REFERENCE

REFERENCE MANUAL

Acknowledgements, Credits and Copyright

S-Base is published by:
Longman Logotron
124 Cambridge Science Park
Milton Road
Cambridge CB4 4ZS
Telephone (0223) 425558

Reference Documentation, S-Base Design & Programming: Simon Glass
Tutorial Documentation, Video & AddressBook Applications: Steve Simatos

S-Base
Copyright © Simon Glass, Steve Simatos & Logotron Ltd. 1992

ISBN 0582 21848 9

Please read carefully before using S-Base:

All rights are reserved. No part of this publication, either the whole or any part of the program or documentation, may be reproduced, transmitted, transcribed, stored in a retrieval system or translated into any language or computer language in any form or by any means electronic, mechanical, optical, chemical, manual or otherwise, without the prior express written permission of Logotron Ltd.

Logotron Ltd. provides the program S-Base under the rules of copyright, and licenses its use to the purchaser. This licence means that, unless you have Logotron's permission to the contrary:

a) you may use the program on a single computer, or move the program and use it on another computer, but you may not use the program on more than one computer at a time;

b) you may make one copy only of the program for backup purposes. Each disc is signed with a unique traceable number;

c) you may not make any copy of the documentation.

Disclaimer

No warranty is made with respect to the program, its quality or performance, its merchantability or fitness for any particular purpose. The program is sold 'as is'. You, the user assume responsibility for the selection of the program to achieve your intended results, and for installation, use and results obtained from the program.

Contents

1. Introduction

Conventions 1
Syntax description 1
Program code 2

2. General concepts

S-Base 3
S 3
Objects 3
Handles 3
Filespec 3
File field 4
Tokenising 4
The Window Manager 4
Polling 4
Disc heaps 4
External file fields 4
Record buffers 5
OS units 5

3. Command reference

archive 7
array delete 8
array insert 9
bit 9
bit clear 10
bit dispose 11
bit new 11
bit set 12
bput 13
buffer clear 13
buffer delete 14
buffer dispose 14
buffer insert 15
buffer load 15
buffer new 17
buffer save 18
buffer set ext 20

card clear protect 20
card clear table 21
card dispose 22
card element box 23
card exit poll 24
card hide 24
card new 25
The card layout 25
Element types 26
Format areas 27
The order of format elements 27
Handlers 27
Nested cards 28
Automatic report generation 29
Evaluation and field width 29
Selectors 30
Tables 31
Icons 31
User control 32
Automatic recalculation 33
The value bar 34
Protected elements 35
Moving the cursor 35
card poll 36
card set cursor 37
card set element 37
card set handle 38
card set protect 39
card set table 40
card show 40
card update 41
card update valuebar 42
case 43
circle 44
clear 45
clg 46
close 46
create 47
def local error 48
def proc 49
delay 51
delete 52

dim 53
draw 55
else 56
endcase 56
enddef 57
endwhile 57
error 57
error set 58
exit 59
file set field 60
flush 61
for 62
ffor 63
gcol 64
if 65
import 66
load 67
local 68
lvar 70
menu clear protect 70
menu clear select 71
menu dispose 71
menu insert 72
menu new 74
menu set 76
menu set menu 77
menu set protect 77
menu set select 78
move 78
next 79
origin 79
oscli 80
otherwise 80
pack 81
plot 81
pmenu dispose 82
pmenu hide 82
pmenu insert 83
pmenu new 84
pmenu poll 86

pmenu show 87
point 87
print 88
proc 90
rectangle 91
reindex 92
repeat 93
report dispose 93
report exit poll 94
report exit run 94
report hide 95
report new 95
Format areas 96
Entry size 98
The order of format elements 99
Text only reports 99
User control 100
Multi-entry reports 100
Automatic report generation 101
report run 102
report set entry 102
report show 104
run 104
save 105
select 106
Filters 107
Record number ordering 108
Updating 108
selector clear protect 110
selector delete 110
selector insert 111
selector set 112
selector set protect 114
set ptr 114
set ext 115
sort 115
sys 117
table clear protect 118
table delete 119
table dispose 119
table exit poll 120

table ext 121
table hide 121
table insert 122
table new 123
The table layout 123
Element types 124
Format areas 125
Table entries 126
The order of format elements 126
Entry size 127
Handlers 127
Nested tables 129
Automatic report generation 129
Evaluation and field width 129
Selectors and Icons 129
User control 129
Automatic recalculation 131
The value bar 131
Protected elements 131
Moving the cursor 131
Redrawing and the cursor 131
Attaching tables to cards 132
Table extension 133
Inserting and deleting entries 133
table poll 134
table set cursor 135
table set element 135
table set entry 136
table set ext 137
table set handle 137
table set protect 138
table show 138
table update 139
table update valuebar 140
then 140
unlock 141
until 141
update 142
use 142
vdu 144
when 145
while 145

zap 146
zero 147

4. Function reference

@abs 149
@acos 149
@addr 149
@adval 151
@altered 151
@asc 152
@asin 153
@atan 153
@atanq 153
@bget 154
@bitfirstset 154
@bitsetcount 155
@bittest 156
@bof 156
@checkmem 157
@chr 158
@clickelement 158
@clickelementlabel 159
@clickentry 160
@clickmouse 161
@cos 162
@cosh 162
@cursor 162
@date 163
@datesi 163
@datesn 164
@deg 164
@deleted 165
@dollar 165
@elaltered 166
@element 167
@elementhandle 168
@elementlabel 169
@elementvalue 170
@entry 171
@entryaltered 172
@entrycount 173

@entrytext 173
@eof 174
@errnum 175
@errstr 175
@eval 176
@exp 177
@ext 177
@externalhandle 178
@filefield 179
@find 179
Updating 180
@findelem 181
@findn 183
@formstr 184
@fpeek 185
@freemem 185
@handle 186
@indexcount 186
@inkey 187
@instr 188
@int 188
@ipeek 189
@justaltered 190
@key 191
@lastentry 191
@left 192
@len 192
@ln 193
@loadindex 193
@lock 194
@log 195
@match 196
@max 197
@mean 197
@menu 198
@menufind 198
@menuselect 199
@mid 200
@min 201
@monotime 202

@msg 202
@mult 203
@next 204
@nextfree 205
@onespace 205
@openin 206
@openout 207
@openup 207
@order 208
@pad 209
@page 210
@pi 211
@point 211
@previous 211
@protectedhandle 212
@ptr 213
@rad 214
@record 214
@recordcount 215
@recsize 215
@right 216
@rnd 216
@round 217
@sdev 217
@selectedhandle 218
@sgn 219
@sin 219
@sinh 219
@spc 220
@sqr 220
@str 221
@string 221
@stringize 222
@sum 223
@tan 224
@tanh 224
@teval 224
@time 225
@timedm 226
@timedom 226

@timedow 227
@timedoy 227
@timefod 227
@timefom 228
@timeform 228
@timefoy 230
@timeh 230
@timemoy 231
@timemy 231
@times 231
@timesi 232
@timesn 232
@timest 233
@timeu 233
@timey 234
@todate 234
@tof 235
@token 235
@tolower 236
@totime 236
@toupper 237
@trim 237
@val 238
@version 238
@vrecsize 239

Appendix A: Key codes **241**

Appendix B: Wimp event codes **245**

Index **247**

1. Introduction

The S-Base Reference is divided into three main sections: General concepts, Command reference and Function reference. You should read and understand the General concepts before looking at the material in the other two sections. The commands and functions are listed in alphabetical order for easy reference. It is not intended that you read this manual from cover to cover.

Conventions

Each command and function is described under the following headings:

Syntax	This gives a formal definition of the command or function syntax. Where there are several different syntaxes available they are numbered and described separately in the text.
Purpose	This describes the general purpose of the command or function.
Arguments	This lists the arguments to the command or function. Each argument is described separately and default values are given where appropriate.
Returns	This section exists only for functions. It first gives the type of the return value, then the meaning of it, possibly referring to the parameters.
Use	This describes how and when to use the command or function. It gives a fuller description of the parameters and the return value (for functions).
Example	This gives a short example of how to use the command or function. Often a *full context* example is given: a complete piece of working S code to illustrate the use of the command or function.
See also	This section refers to other commands and functions which relate to this one in some way. If you need to know more about a particular area of S-Base, you can follow these links to obtain more information.

Syntax description

Under the *syntax* header, the syntax for each commands and functions is

written like this:

selector insert ***handle*** [, ***before***] { , ***string*** }+
@sum (***var*** | ***filefield*** [, ***length***])

S keywords (**selector insert** and **@sum**) are written in bold. These should be typed verbatim into an application.

Parameters to commands and functions are written in bold italics. When entering a command or function into an application these should be replaced by and S expression. The requirements for this expression are listed under *arguments* in the command or function description. For example, ***handle*** in **selector insert** should be replaced by an integer expression which provides the handle of the selector to be operated on.

Optional parts of the syntax are enclosed in square brackets [ ]. Nowhere in the syntax of this manual do these square brackets conflict with square brackets as used by the S language. S uses these for array subscripts, but these do not appear in any syntax in this manual.

In the first example above, ***handle*** is compulsory. Then a comma (,) and ***before*** are optional and can be left out.

Repeated parts of the syntax are enclosed in braces { }. Parts of the syntax with braces can be repeated zero or more times with a comma (,) between each repetition. No comma follows the last repetition. You could leave it out altogether or repeat it 12 times. Where a small plus sign (+) is written after the closing brace, the enclosed syntax must appear at least once.

In the first example above, a comma followed by ***string*** can appear any number of times, but at least once.

Alternative parts of the syntax are listed separated by a vertical bar (|). Exactly one of the possible syntax parts should be included.

In the second example above, either ***var*** or ***filefield*** must following the bracket.

A sequence of lines of S code is shown as an ellipsis (...). This is used to denote any number of lines of code, including none.

Program code

Program fragments are included mostly in the 'Example' section of each command and function. All programs fragments are written in `typewriter font`.

Within the text of this manual, references to example variables and application resources are also written in typewriter font. For example, the variable `num_items`, the `video` file, etc.

2. General concepts

This chapter explains a number of general concepts in a more formal way than might be found in the Tutorial. You may find the descriptions here useful when using this manual. These concepts are not listed in alphabetical order. Throughout this chapter, references to other 'concepts' are written in italics like *this*.

S-Base

S-Base is an application which assists in application development. It provides a number of editors for various parts of an application. It provides a language called *S* which can control S-Base.

S

S is the language within S-Base. It gives the application a high degree control over S-Base. S consists of about 300 commands and functions, variable and *file field* support, and an expression evaluator.

Objects

Within S-Base, an object is an entity created by the application. Cards, tables, reports, buffers and selectors are all objects. S-Base keeps constant track of these objects and automatically destroys them when the application terminated. Commands are provided within S for creating, destroying and using these objects.Each object is identified by a unique *handle*.

Handles

A handle is a positive integer which refers to an *object* in S-Base. All objects are referenced by their handle. S-Base will allocate handles in a random fashion, so you cannot rely on any particular handle being used for a particular object. Handles are generally stored in variables.

Filespec

A filespec is a reference to a file. In S this can be given in three ways:

(1) As a string: the file label. For example `"video"`.

(2) As an integer: the file number. For example `5`.

(3) As a structure: the file structure. For example `video`.

All three are recognised as the same thing in a filespec. The third option is by far the fastest for S-Base to execute.

File field

A filefield is a reference to the field of a file. It is always written as 'file.field'. For example, `video.title` references to the `title` field of the `video` file.

Tokenising

To speed up the execution of application, S-Base processes them before execution. This process is called tokenising and it completely transparent in operation. A few S functions deal with tokenised expressions however, so a function is provided to tokenise an expression.

The Window Manager

S-Base is a client of the RISC OS Window Manager. The Window Manager handle the display of windows and icons, the computer's memory and keyboard and mouse input to a multitasking program. In general S-Base hides the details of the Window Manager from applications. The Window Manager is often called the 'Wimp'.

Polling

The Window Manager requires that all applications *poll* it to find out what it wants the application to do. S-Base handlers all of this for the application through commands such as **card poll**, **report run**, etc. In general, the application is only receptive to user control when polling the wimp. A period of time where the application is not polling the wimp should be marked with an hourglass, using the **delay** command.

Disc heaps

A disc heap is analogous to the computer's memory, but is stored on disc. Chunks of data of any size can reside in the disc heap and can be accessed via an integer block handle. Disc heaps are used by *external file fields*. The **buffer load** and **buffer save** commands are provided to work with external file fields. S-Base uses a disc heap to store parts of the application definition, and for files which contain external file fields.

External file fields

An external file field is 4 bytes long, and contains a block handle. This allows the data associated to the file field to be accessed in the file's disc heap. You can find out the block handle where a file field is stored using **@int** on the file field.

Record buffers

Every disc file in an S-Base application has a record buffer allocated to it. This holds one record from the file. You can only load one record at a time into this buffer. Any reference to a filefield (e.g. `video.title`) will get its value from this record buffer, and place it there for an assignment.

OS units

RISC OS uses a coordinate system called OS units for drawing on the screen. All the S graphics commands and functions use this coordinate system. The origin (0, 0) is at the bottom left of the screen and the top right is often around (1208, 1024) depending on the size of the screen. One OS units can correspond to any number of screen pixels, generally two in the *x* direction and 2 or 4 in the *y* direction.

3. Command reference

archive

Syntax

archive ***handle***, {***value*** [**ext** ***size***] }+

Purpose

To write binary values to a file.

Arguments

handle is the handle of the file to write to (opened with **@openout** for example).

value is a value to write.

size is the number of bytes from ***value*** to write (all by default).

Use

This command allows you to write data to an external file not defined in the file resource. This is useful where you must export data in a form which S-Base cannot generate.

The format used for various value types is as follows:

Type	Format
Integer	Four bytes, least significant first.
Float	Eight bytes in IEEE format.
String	One byte string length (0-255). Then the string itself without a terminating null character.
Time	Four bytes, seconds since 1/1/1970.
Date	Four bytes, days since 1/1/0000 (although there was no year zero, this is used as a base for this data type).
External	The data stored in the disc heap.

Other types cannot be written. If a different format is required you will have to resort to **bput** and **@addr**.

No type information is written to the file, so it will not generally be apparent by reading the file whether a value is an integer or float, for example.

size can only be given with external values. It allows writing of the first

size bytes of an external to a file. If finer control is required the external can be loaded into a buffer first (see **buffer new**).

Example

```
{ open the security lock file and write out the user's
name }
f = @openout ("<Video$Dir>.userlock")
archive f,user.name
close f

| write the first 16 bytes of an external to a file
archive f, pupil.photo ext 16
```

See also

bput, import, @addr, @openout, @openup, set ptr

array delete

Syntax

array delete ***var*** **|** ***filefield*** **[,** ***count*****]**

Purpose

To delete elements from an array.

Arguments

var is a variable array to process. The subscript specifies the start element for deletion (default 0).

filefield is a file field array to process. The subscript specifies the start element for deletion (default 0).

count is the number of elements of the array to delete (default 1).

Use

This command deletes elements from a variable or file field array.

If a subscript is given with ***var*** or ***filefield*** then the deletion will start from that element, otherwise it starts from the first element.

Elements in the array will be moved down to fill the gap and ***count*** elements at the end of the array will be zeroed in this process.

Example

```
dim i test [100]
for i=0 to 99
   test [i] = i * 2
next

| delete elements 50 to 60
array delete test [50], 10
```

```
| insert blank elements before element 50
array insert test [50], 10

| print out all the elements
for i=0 to 99
   print test [i]
next
```

See also

array insert, dim, table delete, @sum

array insert

Syntax

array insert *var* **|** *filefield* **|,** *count***|**

Purpose

To insert elements into an array.

Arguments

var is a variable array to process. The subscript specifies the start element for insertion (default 0).

filefield is a file field array to process. The subscript specifies the start element for insertion (default 0).

count is the number of elements to insert into the array (default 1).

Use

This command inserts zeroed elements into a variable or file field array.

If a subscript is given with ***var*** or ***filefield*** then the new elements will be inserted before that element, otherwise the new elements will be inserted before the first element.

Elements in the array will be moved up to make room for ***count*** zeroed elements and ***count*** elements will fall off the end of the array.

Example

See **array delete** for an example.

See also

array delete, dim, table insert, @sum

bit

Syntax

(1) **bit** ***dst_handle, src1, operator src2***

(2) **bit** ***dst_handle, = src1***

Purpose

To perform bitwise operations on bitstrings.

Arguments

dst_handle is the handle of a bitstring to receive the results.

src1 is the handle of the first source bitstring.

operator is the operator for the operation to be performed (&, |, ~ or =).

src2 is the handle of the second source bitstring.

Use

This command allows operations to be done on bitstrings. Four operations are supported: *bitwise and* (&), *bitwise or* (|), *bitwise exclusive or* (~) and *assignment* (=).

In the case of assignment (using syntax (2)), the contents of ***src1*** are assigned to ***dst_handle***. The original contents of ***dst_handle*** are destroyed.

For the other three operators, the operation is: ***dst_handle = src1 operator src2***. S-Base scans ***src1*** and ***src2*** and performs the operation bit by bit. If ***src1*** and ***src2*** do not contain the same number of bits, then S-Base only scans bits present in both. The original contents of ***dst_handle*** are overwritten by the operation up to the number of bits processed. Bits in ***dst_handle*** after this are unaffected. You can use **bit clear** first if this is a problem.

Example

```
| assign bsrc to bdst
bit bdst, = bsrc

| set bdst to b1 & b2
bit bdst, b1, & b2
```

See also

bit new

bit clear

Syntax

bit clear ***handle*** [, ***start_bit*** [, ***count*** ||

Purpose

To clear bits in a bitstring.

Arguments

handle is the handle of the bitstring.

start_bit is the first bit to clear (0 by default).

count is the number of bits to clear (1 by default if ***start_bit*** is given,

otherwise all).

Use

bit clear clears bits in a bitstring. If ***start_bit*** and ***count*** are not given, the entire bitstring is cleared. Otherwise, ***count*** bits (default 1) starting from ***start_bit*** are cleared.

Example

See **bit new** for an example.

See also

bit new, **bit set**

bit dispose

Syntax

bit dispose ***handle***

Purpose

To dispose of a bitstring after use.

Arguments

handle is the handle of the bitstring to dispose.

Use

bit dispose should be used to dispose of a bitstring after it is finished with. It frees all memory used by the bitstring.

Example

See **bit new** for an example.

See also

bit new

bit new

Syntax

bit new ***handle***

Purpose

To create a new bitstring.

Arguments

handle is an integer variable into which the handle for the new bitstring will be placed.

Use

bit new creates an empty bitstring and returns its handle.

A bitstring is a set of binary flags. Each flag can be on (*set*) or off (*clear*), and the number of flags is limited only by available memory (one byte of memory holds eight flags).

Bitstrings are useful for processing lists of data, recording just a single flag for each item in the list. Selectors use bitstrings for remembering which entries are protected and selected.

Bits within the bitstring can be set and cleared individually. A bitstring has no explicit size, but is always large enough to hold the last 'on' bit in the bitstring. Setting a bit in a bitstring beyond the last set bit will cause the bitstring to expand. Clearing the last set bit will contract the bitstring.

Example

```
! creating a bitstring from the deleted records
bit new b
for i = 1 to @recordcount address
   if @deleted (address, i) then bit set b, i
next
print "Number of deleted records: ", @bitsetcount b
print "First deleted record: ", @bitfirstset b

{ now print out the numbers of all deleted records, and
clear them as we go }
for i = 1 to @recordcount address
   if @bittest (b, i)
      bit clear b, i
      print "Record ",i
   endif
next
```

See also

@bittest, @bitsetcount, @bitfirstset, @bittest

bit set

Syntax

bit set ***handle*** **|,** ***start_bit*** **|,** ***count*** **||**

Purpose

To set bits in a bitstring.

Arguments

handle is the handle of the bitstring.

start_bit is the first bit to set (0 by default).

count is the number of bits to set (1 by default if ***start_bit*** is given, else all).

Use

bit set sets bits in a bitstring. If ***start_bit*** and ***count*** are not given, the entire

bitstring (up to the size of the bitstring) is set. Otherwise, ***count*** bits starting from ***start_bit*** are set.

Example

See **bit new** for an example.

See also

bit clear, **bit new**

bput

Syntax

bput ***handle***, { ***number*** | ***string*** }+

Purpose

To write bytes to a file.

Arguments

handle is the handle of the file to write to (opened with **@openout** for example).

number is a byte to write.

string is a string to write.

Use

This command writes single bytes or strings to a file.

If a number is given, it is written to the file as an 8-bit value. If a string is given, each character of the string is written to the file, without a terminator or length byte.

See also

archive, **set ptr**, **@openout**, **@openup**, **@bget**

buffer clear

Syntax

buffer clear [**ext** | ***handle*** |, ***value*** [, ***start*** |, ***size*** | | |

Purpose

To clear part of a buffer.

Arguments

handle is the buffer to clear.

value is the byte value to fill the buffer with (0 by default).

start is the first byte in the buffer to clear (0 by default).

size is the number of bytes to clear (all by default).

Use

buffer clear is useful for altering the contents of a buffer. The section of

size bytes starting at ***start*** are cleared to ***value***. If **ext** is given, the buffer size is altered to (***start*** + ***size***).

Example

```
| make a buffer containing 1000 zero bytes
buffer new b
buffer clear ext b, 0, 0, 1000

| now set the last 500 bytes to 25
buffer clear b, 25, 500
```

buffer delete

Syntax

buffer delete [**ext**] ***handle***, ***start***, ***size***

Purpose

To delete part of a buffer.

Arguments

handle is the buffer to process.

start is the first byte in the buffer to delete.

size is the number of bytes to delete.

Use

buffer delete is useful for deleting a section of the buffer. The section of ***size*** bytes starting at ***start*** are removed and bytes above are moved down to fill the gap. If **ext** is given, the buffer size is reduced by ***size*** bytes. If not, the ***size*** bytes at the end of the buffer are *not* cleared (use **buffer clear** to do this).

Example

```
{ delete bytes 1000-1499 from the buffer and close up
the gap }
buffer delete ext b, 1000, 500
```

See also

buffer clear, **buffer insert**

buffer dispose

Syntax

buffer dispose ***handle***

Purpose

To dispose of a buffer after use.

Arguments

handle is the handle of the buffer to dispose.

Use

buffer dispose should be used to dispose of a buffer after it is finished with. It frees all memory used by the buffer.

Example

See **buffer new** for an example.

See also

buffer new

buffer insert

Syntax

buffer insert [**ext**] ***handle***, ***start***, ***size***

Purpose

To insert bytes into the middle of a buffer.

Arguments

handle is the buffer to process.

start is the byte in the buffer to insert before.

size is the number of bytes to insert.

Use

buffer insert inserts ***size*** bytes before byte ***start*** in the buffer. If **ext** is given, the buffer size is increased by ***size*** bytes, otherwise bytes will fall off the end of the buffer. The inserted bytes are not necessarily zero (use **buffer clear** to set their value).

Example

```
{ insert 500 zero bytes before byte 1000 in the buffer,
and increase its size to accommodate the change }
buffer insert ext b, 1000, 500
buffer clear b,'0, 1000, 500
```

buffer load

Syntax

(1) **buffer load** [**ext**] ***handle***, ***src_external*** [, ***start*** [, ***size***]]

(2) **buffer load** [**ext**] ***handle***, ***src_handle*** [, ***start*** [, ***size***]]

(3) **buffer load ptr** [**ext**] ***handle***, ***src_address*** [, ***start*** [, ***size***]]

Purpose

To load data into a buffer.

Arguments

handle is the handle of the buffer into which data will be loaded.

src_external is an external file field containing data to load.

src_handle is the handle of an object from which to load data (another buffer or a file).

src_address is a memory address from which to load data.

start is the start position in the buffer for loading the data (default 0).

size is the number of bytes to read into the buffer from the source (all available data by default).

Use

buffer load loads data into a buffer from a variety of sources. If **ext** is given, the buffer will be resized (expanded or contracted) to exactly hold the data to be loaded. Otherwise only data which fits will be loaded.

The most commonly used source is an external file field. S-Base will load the contents of the file field from the disc heap into the buffer. If the file field does not point to a block in the heap block (i.e. **@int** (***src_external***) = -1), no bytes will be loaded. If ***size*** is not given (or is −1), the entire disc heap block will be loaded.

If ***src_handle*** is a file, S-Base will read data from the file at the current file position, updating the file pointer. If ***size*** is not given (or is −1), the rest of the file is loaded in. If there is not enough space between ***start*** and the end of the buffer to load the requested ***size***, then only the amount of data which will fit is loaded. The file pointer will be updated to point just after the last byte read. If you give a ***size*** larger than the remaining number of bytes in the file, S-Base will only read what is available.

If ***src_handle*** is another buffer, S-Base will read data from that buffer starting at its start. If ***size*** is not given (or is −1), the entire contents of the source buffer will be read in, provided there is sufficient space in ***handle***. This variation on **buffer load** allows you to rapidly copy data between buffers, and so perform fairly complex operations on the data quickly. For example, you could break a sprite file into separate files containing one sprite each by loading the file into a buffer and copying parts of it into another buffer for writing to a file.

buffer load ptr is a variation on this command which loads an arbitrary memory block into a buffer. ***src_address*** must be a valid memory address, generally obtained using **@addr**. If an invalid address is given, a fatal error will be reported. Note that since **buffer load ptr ext** may extend or contract the buffer causing objects in memory to move, ***src_address*** may

be out of date when S-Base actually loads the memory into the buffer. For this reason, you should normally extend the buffer using **buffer set ext** before loading the data. **buffer load ptr** is mainly useful for copying data from an integer array after processing.

buffer new

Syntax

buffer new [**static**] ***handle***

Purpose

To create a new buffer.

Arguments

handle is an integer variable to receive the buffer handle.

Use

buffer new creates an empty buffer and returns its handle. If **static** is given, then the buffer is created in 'stable' memory. That is, the address of the buffer will not move once set (unless its size is changed). Stable buffers should only be used in special circumstances since when they are destroyed, their buffer memory cannot always be returned to the Operating System.

A buffer is simply a block of memory of variable size. A buffer can hold a sprite, a drawing, a section of a file read from disc, etc. Buffers are limited in size by available memory. You can have any number of buffers, but you must remember the handle of each so it can be accessed.

A buffer's size is measured in bytes, and within a buffer bytes are numbered from 0. The first byte is byte 0, the last is byte (**@ext** ***handle*** – 1).

Buffers are useful for processing external data, since you can load a buffer with the contents of an external file field, alter it and save it back. Buffers can also be used for general purpose file processing, since you can read a section of a file into a buffer and write it back to another file.

Example

This procedure assumes a directory 'photos' containing 20 sprite files named '1' to '20'. The sprites are loaded in and placed into the first 20 records of a `pupil` file. Note that `pupil.photo` is an external field. Sprite files start with a 12 byte header which is ignored in this procedure.

```
def proc import_photos
local fd, b, rec

| create a buffer to hold the photo
```

```
buffer new b

for rec = 1 to 20
   | open up the file containing the photo
   fd = @openin ("<S_Appl$Dir>.photos."+@str (rec))

   { skip the 12-byte header at the start of the sprite
file }
   set ptr fd, 12

   { load the rest of the file into the buffer,
extending the buffer as required }
   buffer load ext b, fd

   | load the pupil
   load pupil, rec

   { save the buffer into the external file field,
replacing whatever was there before }
   buffer save b, pupil.photo

   | write back the pupil record
   update pupil, rec

   | & close the file
   close fd
next

| destroy the buffer
buffer dispose b
enddef
```

buffer save

Syntax

(1) **buffer save** ***handle***, ***dst_external*** [, ***start*** [, ***size***]]
(2) **buffer save** ***handle***, ***dst_handle*** [, ***start*** [, ***size***]]
(3) **buffer save ptr** ***handle***, ***dst_address*** [, ***start*** [, ***size***]]

Purpose

To save data from a buffer.

Arguments

handle is the handle of the buffer from which data will be saved.

dst_external is an external file field to receive the data.

dst_handle is the handle of an object to save the data to (another buffer or a file).

dst_address is a memory address to which data should be written.

start is the start position in the buffer for saving the data (default 0).

size is the number of bytes to write from the buffer to the destination (all available data by default).

Use

buffer save saves data from a buffer to a variety of destinations.

The most commonly used destination is an external file field. S-Base will save the contents of the buffer into the disc heap and update the file field to point to the new heap block. Any old heap block pointed to by the external will be freed first. If ***size*** is not given (or is −1), the entire buffer starting from ***start*** will be saved. Otherwise, only ***size*** bytes starting at ***start*** will be written. If you write zero bytes to an external file field no new heap block will be created and the existing block will be freed. In this case **@int** (***dst_external***) will then return −1.

If ***dst_handle*** is a file, S-Base will write data to the file at the current file position, updating the file pointer. If ***size*** is not given (or is −1), the rest of the buffer (from ***start***) is saved. If you give a ***size*** larger than the remaining number of bytes in the buffer, S-Base will only write what is available.

If ***dst_handle*** is another buffer, S-Base will write data to that buffer starting at the start. If ***size*** is not given (or is −1), the entire contents of the source buffer will be written starting from ***start***, provided there is sufficient space in ***dst_handle***. This variation on **buffer save** allows you to rapidly copy data between buffers, and so perform fairly complex operations on the data quickly.

buffer save ptr is a variation on this command which allows you to save the contents of a buffer to an arbitrary memory address. ***dst_address*** must be a valid memory address, generally obtained using **@addr**. If an invalid address is given, a fatal error will be reported. Note that writing data into a memory address can be very dangerous. If an incorrect operation is performed, you may overwrite S-Base workspace and crash the application. **buffer save ptr** is mainly useful for copying data to an integer array for processing.

buffer set ext

Syntax

buffer set ext ***handle, size***

Purpose

To alter the size of a buffer.

Arguments

handle is the handle of the buffer to alter.

size is the new size of the buffer.

Use

buffer set ext sets the size of a buffer to ***size***, by either expanding or contracting it. It is useful for resizing a buffer before calling **buffer load**.

Example

```
| create a 1000 byte buffer
buffer new b
buffer set ext b, 1000
```

See also

buffer new, **@ext**

card clear protect

Syntax

card clear protect ***handle***, {***element***}$^+$

Purpose

To unprotect elements of a card.

Arguments

handle is the handle of the card to alter.

element is an element to unprotect.

Use

This command allows you to unprotect elements in response to user action.

Unprotected elements can be clicked on and can contain the cursor. See **card new** for more details.

Elements protected using the 'format element' flags dialogue can still be unprotected with this command.

Example

This example is similar to the example for **card set protect**. This is the click handler for the record browser. When the user is looking at the first record the left arrow is protected; when looking at the last record the right arrow is protected.

```
| remember whether we need to update or not
local up = 0

case @clickelementlabel
when "left"
   if recnum > 1
      recnum -= 1
      load curfile, recnum
      up = TRUE
   endif
when "right"
   if recnum < @recordcount curfile
      recnum = recnum + 1
      load curfile, recnum
      up = TRUE
   endif

| update the card if necessary
if up
   card update c

| protect fields as necessary
   if recnum = 1
      card set protect c,"left"
   else
      card clear protect c,"left"
   endif

   if recnum = @recordcount curfile
      card set protect c,"right"
   else
      card clear protect c,"right"
   endif
endif
```

See also

card clear protect, card new

card clear table

Syntax

card clear table ***handle, element***

Purpose

To detach a table from a card.

Arguments

handle is the handle of the card to remove the table from.

element is the element in the card to which the table is currently attached.

Use

card clear table detaches a table previously attached to a card with **card set table**. The table can then be disposed of in the normal way with **table dispose**.

Example

See **table new** for an example.

See also

card set table, table new, card dispose

card dispose

Syntax

card dispose [update] ***handle***

Purpose

To dispose of a card after use.

Arguments

handle is the handle of the card to dispose.

Use

This command should be used to dispose of a card after it is finished with. It first does a **card hide**, then removes the card from memory.

Any selectors attached to the card are also disposed of. Any tables attached to the table are detached with **card clear table**.

If **update** is given in the command, the template used to create the window is updated with the card's position when last hidden. This means that the card will always pop up where the user last left it rather than returning to its original position every time it is recreated.

Cards must be disposed of after use; it is not advisable simply to hide them from view.

Example

See **card new** for an example.

See also

card new

card element box

Syntax

card element box ***handle, element, array***

Purpose

To find the position of a card element on screen within a redraw handler.

Arguments

handle is the handle of the card to use in the calculation.

element is the element to find the position of.

array is an integer-4 array with at least four element to receive the results.

Use

This command is only useful in redraw handlers associated with cards. It works out the screen position of an element and puts it into the array. The coordinates are stored in the order x0, y0, x1, y1.

You can use these coordinates in normal graphics commands to draw over an element. Often the element exists only to mark the position where an object should be drawn (put the element in the 'special' area in this case).

Redraw handlers should be used only where it is not possible to achieve the desired effect with normal S-Base elements like sprites and drawings. The handler will be called whenever S-Base wants any part of the card to be redrawn. The handler is called *after* S-Base has finished redrawing, so it will appear over the top of parts of the card drawn by S.

Example

This example draws a circle over the top of an element called `circle` in a card:

```
local a
dim i a [4]

| find the position on screen of the 'circle' element
card element box c, "circle", a

| set the colour (red)
gcol 11

| draw a filled circle within the element
circle fill (a [0] + a [2]) / 2, (a [1] + a [3]) / 2, (a
[2] - a [0]) / 2
```

card exit poll

Syntax

card exit poll ***handle***

Purpose

To stop polling a card.

Arguments

handle is the handle of the card to stop polling.

Use

This command allows control to return from an executing **card poll** command.

S-Base marks the card ***handle*** as completed. If S-Base is currently waiting inside a **card poll** command for that card, then the **card poll** command will return and execution will continue from that point.

Note that this command has no immediately discernible effect if S-Base is currently polling another card or table, etc., even if that card or table's **card poll** is nested inside the **card poll** for card ***handle***. You should only use **card exit poll** on a card when you know that S-Base is currently executing a **card poll** for that card. **card poll** will execute until told to stop by a **card exit poll** for the same card.

Example

```
| click handler for a browser card

| check what the user clicked on
case @clickelementlabel

| if quit, then quit the card
when "quit"
   card exit poll @handle

endcase
```

See also

card poll

card hide

Syntax

card hide ***handle***

Purpose

To remove a card from the screen.

Arguments

handle is the handle of the card to remove.

Use

This command should be used after the user has finished with a card. It closes and deletes the window containing the card, removing it from the screen. Any windows attached to the card (selectors and tables) are also closed.

Note that the card will not disappear until the wimp is next polled.

Example

See **card new** for an example.

See also

card new

card new

Syntax

card new ***handle,format***

Purpose

To create a new card.

Arguments

handle is an integer variable into which the handle for the new card will be placed.

format is the format to use for the card. This can be a string (format label) or an integer (a format number) but a format label is to be preferred.

Use

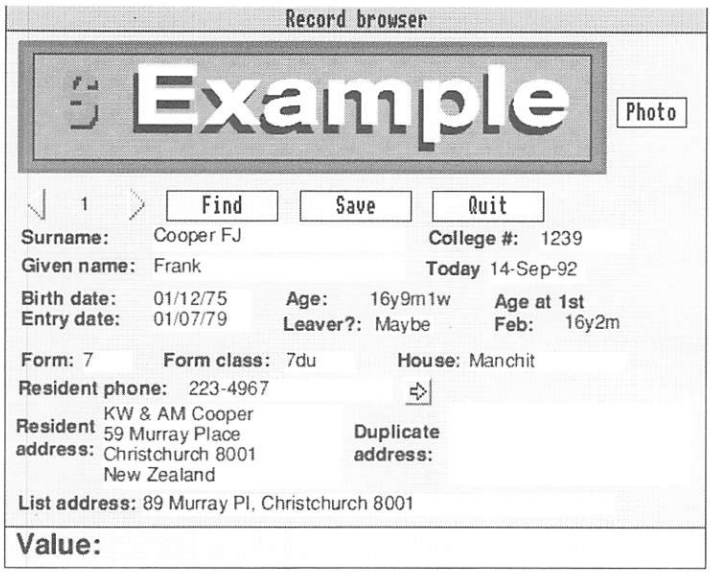
This command is used to create a new card ready for display or entry of data. A card is simply a window containing fields where the user can enter data, as the example below shows.

The card layout

Each card has a format which defines the look of the card. For example, it specifies the elements which should appear in the card and the template, etc. to use for the card. You normally define the look of the window and any icons it has using the template editor. You then attach the template to the format and set out the elements you want to appear in the window.

Cards can also have handlers attached to them. For example, you could instruct S-Base to call a handler every time the user clicks on a certain icon. The 'Quit' icon, for example, would have a click handler which simply exits the card. When the user clicks on 'Quit' S-Base will call your handler and exit the card.

A typical card is shown below. There is a drawing at the top, a row of icons below that and a number of prompts and fields in the main body of the card.



You will normally create a template for your card which defines the look of the window and any icons the window contains. In this example, the window has scroll bars, quit boxes, etc. turned off. There is a row of icons allowing the user to control the browser, and an icon further down for popping up further information.

Element types

Cards support all types of elements you can create in your format:

prompt	The expression is evaluated and displayed in the card.
icon	The expression is evaluated (if there is one) and used to decide whether the icon should be selected or not. The icon is displayed.
field	The expression is evaluated and displayed in the card. The user may move the cursor around the fields and edit them.
selector	A selector window is created and displayed within the card. The selector is a separate entity from the card, but attached to it.

drawing	The drawing is displayed in the card.
sprite	The sprite is displayed in the card.
area	Ignored.
line	Displayed in the card.
rectangle	Displayed in the card.

Format areas

For the most part cards ignore the area of elements (but see **The value bar** below). You would normally place all elements in the 'Page' area. There is no need to create a page area element. All area elements are ignored and are not displayed in the card.

The order of format elements

The format layout editor, like all other places where formats are displayed, displays format elements in order from first to last. This means that for overlapping elements, the first one will be underneath the last one. Unfortunately icons always appear on top of all other elements, as these are drawn by the Window Manager.

Handlers

There is no built-in user control of cards. You must build this by adding click handlers to icons, keypress handlers to fields, etc. Once you have issued a **card poll** command for a card, that command will not return control until **card exit poll** is issued in one of the card's handlers.

S-Base supports the following handlers for cards open on the screen. You will normally use **@element** or **@elementlabel** to decide which element the event relates to:

click	Called when the user clicks in the card. Use **@clickelement** and **@clickelementlabel** to find out what the user clicked on.
keypress	Called when the user presses a key in the card. Use **@key** to get the wimp key code (e.g. 13 is <return>). A list of key codes is given in Appendix A.
enter field	Called when the cursor enters a new field.
exit field	Called when the cursor leaves a field.
menu maker	Called when the user clicks Menu in a card. Use **@clickelement** and **@clickelementlabel** to find out

which element the user clicked Menu over.

menu select Called when the user selects an item from the card menu. Use **@clickelement** and **@clickelementlabel** to find out which element the user clicked Menu over, and **@menuselect** to find out which menu option was selected.

You may find the functions **@justaltered**, **@elaltered** and **@altered** useful in some of these handlers. For example, if you have a search field whose keypress handler searches on whatever the user types you should use **@justaltered**. This tells you whether the key just pressed altered the field. If not, there is no point in performing the search. If you did not take advantage of this optimisation, a time-consuming search would be performed every time the user presses a cursor key!

@altered is most useful when you have closed the card. If anything in the card has been altered, **@altered** will return 1. You need not bother to save the information if **@altered** returns 0. **@elaltered** is only used with tables.

Nested cards

Sometimes you need to open a new card while an existing card is open. You can do this by issuing **card new** and **card show** commands as usual. The original **card poll** will suffice to ensure that the new card operates correctly. Of course, a **card exit poll** command on this new card will not have any affect since that card is not being polled. You will have to hide and dispose of the card where necessary.

Sometimes the new card is displayed as a result of an option invoked from the main card. In this case you want the user to enter information in the new card and click OK or CANCEL, whereupon you close this card and perhaps update the main card in some way. If the user clicks on the main card while the new card is open you want to close the new card. In this case you should issue a **card poll** command for the *new* card, with the **close** option given. This will ensure that the new card remains on the screen only as long as the user doesn't click outside it. As an example:

```
| make the main card
card new main,"format1"
card show main
card poll main
card dispose main
```

The following code may appear in a click handler for the main card:

```
card new sub,"format2"
card show sub
card poll sub, close "cancel"
card dispose sub
```

Here the sub card in the main card's click handler is being polled. When the user clicks outside the sub card, or clicks on the 'Cancel' button, the card is closed.

If there was no **close** option given in the **card poll** for a card, or if the card does not have its own **card poll** command, S-Base will call the card's click handler with **@clickelementlabel** set to `"<scared>"`. You can choose to ignore this, or you may like to close the card.

Automatic report generation

You will normally find automatic report generation adequate for creating simple cards. It can also be useful for creating the basis for a complex card. You should find that all you need to do is write click handlers for all the icons and place the usual **card new**...**card dispose** commands somewhere in your application.

You can use the format layout editor to add lines and rectangles to your cards to make them look more presentable. Logos can be added as sprites or drawings.

See the Tutorial for more information on automatic format generation.

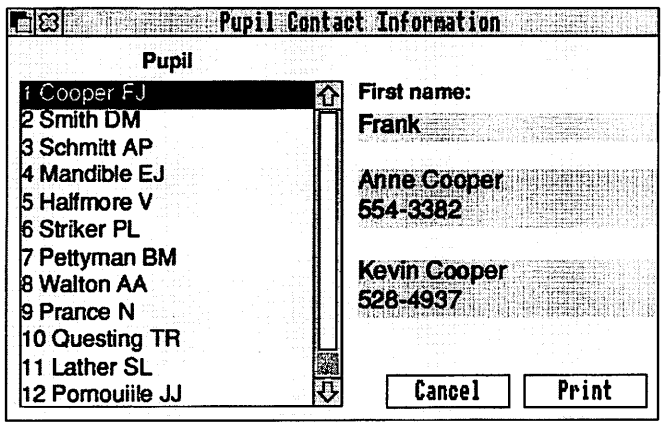
Evaluation and field width

Note that S-Base evaluates all expressions in prompts, fields, icons, etc. when the card is created (**card new**) and stores the values produced. This allows S-Base to determine the number of characters in each field. For example, evaluating `client_code:-4` where client code is 15 will produce `'15  '` and S-Base will set the field length to four characters. Any attempt to enlarge the field after creation (even with **card update**) will fail. S-Base ensures the length always remains the same. You should be careful that fields (particularly numeric ones) are followed by a reasonable field width specifier.

Another effect of this early evaluation is that the values which will initially appear in the card are determined during execution of the **card new** command. Changes to variables or file fields which are 'behind' the card *after* the **card new** command will not appear when the card is displayed. **card update** can be used to re-evaluate particular fields or icons, or even whole cards.

A further point concerns the length of fields using proportionally-spaced fonts. Since the length is fixed during processing of the **card new** command, the user cannot enter more characters into a field than its length allows. However, the space given to the field may be too large for the characters within it. For example, if the user enters 30 dots (.....) into a field which is wide enough for 30 ampersands (&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&), the cursor will not reach even half-way across the field before it becomes full. The user must realise that the field is limited by the number of characters, not the width of those characters. This problem does not occur with monospaced fonts, such as System Font.

Selectors



Any selectors present in the format will be created during processing of the **card new** command. The selectors will be empty and is it up to you to add any items required to the selector before issuing the **card show** command.

A selector is an autonomous object just like the card it is attached to. You can use **@elementhandle** to obtain the handle of the selector, then use the **selector ...** commands to manipulate the selector. Selectors are displayed as scrolling panes within the card.

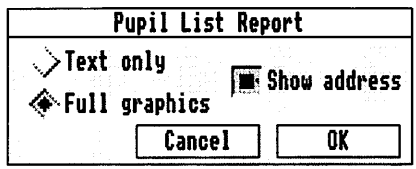
S-Base will ensure that a **card show** command will display the selector and **card hide** will hide it. The selector is disposed of during the **card dispose** command. There is no way to explicitly dispose of a selector while it is still required by the card. You can have any number of selectors attached to a card — each will have its own handle.

Tables

You can attach tables to cards. This is described under the **table new** command. You must create a format area element in the card's format and put it in the *special* area. The table will then be attached to that element of the card and will appear over it in a similar way to selectors.

Like selectors, tables are separate objects. You should not dispose of a table before it has been detached from the card.

Icons



Icons in a format can be 'dumb', in which case they appear, but clicking on them has no effect. However, icons can also operate in a similar way to fields. A single icon can hold the value 0 or 1 (like the 'Show address' icon above). The value 0 means the icon is not selected, while 1 means the icon *is* selected. The 'Show address' icon above has the value 1. Icons can have expressions attached to them just like fields. For example, the above icon uses the expression `show_address`. When the user clicks on the icon, S-Base automatically updates the variable `show_address` to either 0 or 1.

Icon such as this are useful for implementing on/off or yes/no options. Once you have set up the format icon with the correct variable in its expression, you need only set up the variable before calling **card new** and check it when the user clicks OK. S-Base will ensure that the variable has the up-to-date value of the icon.

Simple on/off icons are useful, but it is often necessary to allow the user to select between several possible options. The 'Text only' and 'Full graphics' icons above work like this. Both are in the the same *exclusive selection group* (ESG), so only one can be selected at a time. You can put the same variable in the expression for both icons but give them a different *value*. When one icon is selected, S-Base will set the variable to one value, and S-Base will set the variable to another value when the other icon is selected. You can have any number of icons operate in this way.

Multi-select icons must have the 'value' option turned on in their 'format element info' dialogue, and a valid integer value must be entered. Once this is done, you need only set up the variable before calling **card new** and

check it when the user clicks OK. S-Base will ensure that the variable has the value of the selected icon. If no icon in the group is selected, S-Base will use the last selected icon for the value, so you can be sure that a valid value is always available. Up to 15 ESGs are supported by the Window Manager, so you can have 15 sets of icons grouped in this way.

User control

The user has some control over the card once it is on the screen. The user may move between fields using the mouse and the cursor keys. The order of fields is defined by the *physical* order of elements in the format. So that cursor movement operates intuitively, you should ensure that format elements are 'height sorted', that is, in order of position down the format. There is a height sort option in the format layout editor.

The cursor position is measured in characters, with 0 being the start of a field. The position is always *between* characters. Thus 0 is before the first character and 20 is after the last character for a field of length 20. The cursor may not move outside the range [0...length of field].

S-Base will endeavour to keep the cursor on the screen at all times. Single line fields which have too many characters to display within the space given will scroll left to right to ensure the cursor is kept within the field.

Multi-line fields will scroll top to bottom and are word-wrapped, so S-Base will not break words unless it has to.

S-Base has no 'overwrite' mode. All characters entered into a field are inserted at the cursor position. This may cause characters to drop off the end if the field is already full. When the user moves into a new field and presses a key to insert a character, the old contents of the field are cleared automatically. To avoid this, the user should press the left arrow key first.

The *current element* is the element containing the cursor. This element is returned by **@element** and **@elementlabel**. If the cursor is not in any element (perhaps because all elements are protected), then **@element** will return –1.

The following keys are understood and processed by S-Base before the keypress handler is called:

Key	Action
right arrow	Move cursor to the right within the current field.
left arrow	Move cursor to the left within the current field.
up arrow	Move cursor to the previous line within the

	current field. If already on the top (or only) line, move to the start of the nearest field above the current one.
down arrow	Move cursor to the next line within the current field. If already on the bottom (or only) line, move to the start of the nearest field below the current one.
return	Move to the next field. If already in the last field, move to the first field.
tab	Move to the next field. If already in the last field, move to the first field.
sh-tab	Move to the previous field. If already in the first field, move to the last field.
ctl-U	Erase the current field and move the cursor to the start of the field.
ctl-left arrow	Move the cursor to the start of the current field.
ctl-right-arrow	Move the cursor to after the last non-blank character in the current field.
delete	Delete the character to the left of the cursor and move the cursor back one.
copy	Delete the character to the right of the cursor.

If the card has a close icon and the user clicks on it, S-Base will call the format's click handler with **@clickelement** set to –2 and **@clickelementlabel** to `"<close>"`.

You may wish to check for **@clickelementlabel** being `"<scared>"` in the card's click handler and issue a **card exit poll** to quit the card. S-Base simulates a click on this 'element' when the user clicks outside the card.

Within any handler, if there is no current field, **@element** will return –1 and **@elementlabel** will return an empty string. **@clickelement** and **@clickelementlabel** work the same way.

Automatic recalculation

The expressions you put in fields are evaluated by S-Base and placed on the screen. Thus expressions are used to create the information the user sees. Expressions are also used to return the information the user enters.

For example, evaluating the expression '`title.description`' in the Video database will produce a string which is the video title's description. As the user edits this description, S-Base automatically updates the '`title.description`' field. After every keypress, S-Base re-scans the information entered and converts it into a value. The value is shown in the value bar if present, and also placed back into the expression given.

This process is called *automatic recalculation*. You need only place the card on the screen and S-Base will automatically update the variables and file fields mentioned in the card with anything the user types. Automatic recalculation can be turned off for a particular field. Indeed it must be turned off if the field is anything more than a simple variable or file field. For example, a field containing the expression '`title.location + 1`' will not work with automatic recalculation. Say the user types in the value 6. S-Base knows that it must assign the value '6' to '`title.location + 1`', but is not smart enough to realise that this means it should set '`title.location`' to 5. Automatic recalculation only works with simple fields. In this case you would have to set up a keypress handler for the field, which uses **@elementvalue** to get the value of the field (here 6), subtract 1 and assign that value to '`title.location`'.

A simple test which will help you decide whether S-Base can perform automatic recalculation or not is to think of the expression followed by an equals sign (=) followed by a value. If this is a valid S-Base assignment statement, then all is well. In the example, '`title.location + 1 = 6`' is not a valid assignment statement, so automatic recalculation will not work.

The value bar

Cards can have a value bar attached below them. The value bar displays the value, as determined by S, of the current field. As the user types, the value bar updates.

The value bar helps the user enter values where the result is not immediately obvious. For example, a value bar can make it much easier to enter a time or date. Rather than having to exit the field to see what S-Base has made of their efforts, the user can glance at the value bar.

When the current field is not automatically recalculated (see the Auto-recalculation flag in the 'format element info' dialogue), the value bar is even more useful. The application can take complete control of the contents of the value bar. Since S-Base is not recalculating the field, the application must do this, and update the value bar accordingly. For

example, a *record search* card may allow the user to enter a video title. As the user types in the title the application can search on the value in the field so far (obtained using **@elementvalue**) and display the closest matching title in the value bar using the **card update valuebar** command.

The presence of a value bar in a card is defined by the format. If there is a format element in the 'Special' area with the label `valuebar` then S-Base uses this to create a value bar below the card. The `valuebar` field's font style gives the height and colour of the value bar (the width is the same as the card) and the expression in the field is evaluated to get the text in the value bar. In the example card above, the expression is `"Value: "`.

Protected elements

The user can generally enter any field and change it. Sometimes however, it is necessary to protect a field from alteration. This is done by setting the 'protected' flag in the format field element, or (at run-time) by use of the **card set protect** command.

A protected field cannot be entered either by cursor movement or by clicking in the field. Clicking over a protected element will be ignored (the element's click handler will not be called), as will clicking Menu on the element (the menu maker handler will not be called). You can unprotect a field using the **card clear protect** command.

Fields can be displayed with a different font style when protected. S-Base adds '`.protect`' to the field's usual font style and uses that font style (if found) to display the field when protected. For example, a field using the font style '`normal field`' will use the font style '`normal field.protect`' when protected. Generally the protected font style will be a greyed-out version of the unprotected font style.

You can use protected fields instead of prompts if you wish, and this is the convention when the prompt contains an expression rather than a straight string, since prompts are not updated by the **card update** command.

All types of elements can be protected, but elements such as rectangles and sprites are not displayed differently when protected. Protected icons are displayed 'greyed out' and as with all elements the user will not be able to click on them.

Moving the cursor

S-Base provides commands to allow the application to move the cursor within a field, or to a different field altogether (**card set cursor** and **card set element**). As with all such movement, S-Base notes the change but will

not get around to moving the cursor until the machine is next idle.

Example

```
card new c, "pupil browser"
card show c
card poll c
card dispose c
```

See also

card dispose

card poll

Syntax

card poll [**until** ***eventcode*** ,] ***handle*** [, **close** ***elementspec***]

Purpose

To wait for the user to finish with a card.

Arguments

eventcode is the Window Manager event code to wait for before returning control to the program.

handle is the card to wait for.

elementspec is the element to 'click on' when the card is compromised by a foreign event.

Use

This command is useful after displaying a card on the screen. It waits for a signal that the card is complete (e.g. the **card exit poll** command), then returns control to the program.

If ***eventcode*** is given, then S-Base will cease polling when it sees an event of that type (e.g. 0 for null events) for any window. This is most useful to allow primitive multitasking during monotasking operations: polling until a null event occurs allows other applications to get a look in.

If ***elementspec*** is given, the card will become a *scared* card. This means S-Base will cease polling the card when it detects a mouse click or keypress in another S-Base window. Some cards cannot handle this, and must close. S-Base will call the click handler for the given element, to simulate a click on it, before processing the possibly harmful event. It is usual to make ***elementspec*** the card's 'cancel' button.

Example

See **card new** for an example.

See also

card new

card set cursor

Syntax

card set cursor ***handle, position***

Purpose

To set the cursor position in the current field.

Arguments

handle is the handle of the card.

position is an integer giving the new position, 0 being the extreme left.

Use

This command is used mainly for setting the cursor to the start of a field after updating it. S-Base notes the change and will move the cursor when the machine is next idle.

The cursor position is measured in characters, with 0 being the start of a field. The position is always *between* characters. Thus 0 is before the first character and 20 is after the last character for a field of length 20. The cursor may not move outside the range [0...length of field]. S-Base will ignore any such request.

Example

```
| update the 'name' field and reset the cursor
card update c, "name"
card set cursor c,0
```

See also

card set element

card set element

Syntax

card set element ***handle, element***

Purpose

To change the field containing the cursor.

Arguments

handle is the handle of the card.

element is the element which should gain the cursor.

Use

This command is useful when the cursor must be placed in a particular card, or must be moved around in a card. The cursor position within the element will be set to 0.

S-Base notes the change and will move the cursor when the machine is

next idle.

Example

```
{ open a new card and put the cursor in the 'name'
element }
card new c, "title"
card show c
card set element c,"name"
card poll c
card dispose c
```

See also

card set cursor

card set handle

Syntax

card set handle ***handle***

Purpose

To place the cursor in a card.

Arguments

handle is the handle of the card to receive the cursor.

Use

card set handle is useful when you want a particular card to claim the cursor, perhaps because the user has clicked on a particular icon. S will place the cursor in the card (without changing that card's current element or cursor position) when the machine is next idle.

S does a **card set handle** automatically in the **card poll** command.

Example

```
{ click handler for a card's 'comment field' popup.
c_comment is the handle of the card containing the
comment field, -1 if this card is not yet open. This
handler opens the command card and puts the cursor in it
}

| if the card is not yet created, create it
if c_comment = -1
   card new c_comment, "comment field"
   card show c_comment
endif

| put the cursor in the comment card
card set handle c_comment
```

See also

card set cursor, card set element, table set handle

card set protect

Syntax

card set protect ***handle***, {***element***}+

Purpose

To protect elements of a card.

Arguments

handle is the handle of the card to alter.

element is an element to protect.

Use

This command allows you to protect one or more elements in response to user action. Any element type in a card can be protected. For permanently protected elements you should set the *protected* flag in the format element dialogue.

Protected elements cannot be clicked on and cannot contain the cursor. See **card new** for more details.

Example

In this example a normal record browser card is created. It has arrow icons called 'left' and 'right' which the user can click on to browse between records. These icons are protected if there is only one record in the file.

```
| create a browser card

card new c,"browser"

{ protect the record # arrows if there is only one
record }
if @recordcount curfile < 2
   card set protect c,"left","right"
endif

card show c
card poll c
card dispose c
```

See also

card clear protect, card new

card set table

Syntax

card set table ***handle, element, table_handle***

Purpose

To attach a table to a card.

Arguments

handle is the handle of the card to attach the table to.

element is an element in the card over which the table will appear.

table_handle is the handle of the table to attach.

Use

card set table is useful where you want a table attached to a card. Rather than using **table show** to display the table, you attach the table to the card and then use **card show**. The table will be attached to the card over the ***element*** element (like a selector), which should be an area element in the *special* area.

You must not dispose of a table which is attached to a card. Use **card clear table** or dispose of the card first.

Example

See **table new** for an example.

See also

card clear table, table new

card show

Syntax

card show [static | menu] ***handle***

Purpose

To display a card in a window on the screen.

Arguments

handle is the handle of the card to display.

Use

This command should be used after creating a card. It creates and opens a window containing the card on the screen. Any objects attached to the card (e.g. selectors) will be displayed on top of the card.

A card can be opened in two ways: as a static window or a menu. S-Base will normally choose the correct method by looking at previous Window Manager events and by seeing if anything is attached to the card, but you can override this by putting either **static** or **menu** after the **card show**.

A 'static' card can have selectors, tables and popup menus attached to it. A 'menu' card is opened as a menu or submenu, and will close as soon as another menu is opened. You should normally use static cards. Menu cards are useful when you want a card to be attached to a menu.

A common use for **card show static** is where a card is opened from a menu, but should be opened as a static card. This is used throughout the example Video application.

Note that the card will not appear until the wimp is next polled (e.g. with **card poll**).

Example

See **card new** for an example

See also

card new

card update

Syntax

card update *handle*, {*element*}+

Purpose

To recalculate and re-display elements of a card.

Arguments

handle is the handle of the card to update.

element is an element of the card which needs updating. An element may be given as either the element label or the element number.

Use

This command is useful when the application changes the values 'behind' the card. For example, a record browser which displays a single record from a file will have to update the card when a new record is loaded.

If no elements are given, all the fields and icons in the card are updated and redrawn. Note that S-Base will not update other element types in this situation because this is seldom required. To update a selector or a sprite the element must be explicitly listed in the **card update** command. S-Base will not update these unless specifically told to because of the time penalty involved.

You cannot update prompts with **card update**. Use a protected field instead.

Note that S-Base will only redraw fields which have changed. For other element types, S-Base assumes they have changed and always redraws them.

Example

```
| click handler for right arrow in browser
if recnum < @recordcount curfile
   | load the next record and update the card
   recnum += 1
   load curfile, recnum
   card update @handle
endif
```

card update valuebar

Syntax

card update valuebar ***handle*** [, ***expr*** [***fieldspec***]]

Purpose

To display an updated value in the value bar.

Arguments

handle is the handle of the card to update.

expr specifies an expression to be converted to a string and displayed. If this is omitted, the field's normal expression is used.

fieldspec specifies how the string conversion will take place (see **print**).

Use

This command is useful when the application wants the value bar to show a value for elements which do not have auto-recalculation.

S-Base can never scan a *manual-recalc* field to produce a value. Therefore, when the cursor is in a manual-recalc field, the value bar will be blank. The application may insert its own value using this command.

As an example, imagine a `find record` card which allows the user to enter a name to find. As the user enters a name, the application could search for a matching value and put it in the value bar.

If no expression is given, this command has no effect on manual-recalc fields. For auto-recalc fields, S-Base just updates the value in the value bar (as would normally happen anyway).

Example

```
| keypress handler for 'find title' card

| if the user has just altered the field
if @justaltered
   | search for a title matching what they have entered
   rec = @findn (update title, 0, @trim @elementvalue
(@handle, @element))
```

```
   { if we found one, load it and update the value bar
with the title }
   if rec
      card update valuebar @handle, title.title
   endif
endif

! quit the card when the user presses <return>
if @key = 13 then card exit poll @handle
```

case

Syntax

case ***testexpr***
{ **when** ***expr*** {, ***expr*** }
...}
[**otherwise**
...]
endcase

Note that the **when** commands are *not* separated by commas as the { } suggests.

Purpose

To execute one of a number of possible statements depending on an expression.

Arguments

testexpr is the expression to be matched against all the **when** statements.

expr is an expression giving one option for execution.

Use

This command allows execution to branch to a number of different possibilities depending on the supplied expression ***testexpr***.

When S-Base sees the **case** statement, it evaluates ***testexpr*** and searches through all the following **when** statements for an ***expr*** with the same value. If a match is found, the statements following this **when** are executed until either **when**, **otherwise** or **endcase** is encountered. Execution then continues after the **endcase**.

If no match is found (i.e. no ***expr*** matches ***testexpr***), S-Base executes the statements following the **otherwise** command, if this is present. If there is no **otherwise** and no match is found, no part of the **case** statement will be executed.

Example

```
| normal use of case, with constants after each 'when'
case type
when 1
  print "Type 1 found"
when 2,3,4
  print "Type 2-4 found"
when 5,6
  print "Type 5/6 found"
otherwise
  print "Unknown type"
endcase
```

This example illustrates that the **case** statement can be turned around the other way. Here, we are looking for the first match in a number of possibilities. Note that only one of the **print** statements can be executed. They are ordered from most to least extreme, since if `proc is_way_overdue` returns TRUE then so will `proc is_overdue`.

```
case TRUE
when proc is_way_overdue
  print "Police to be called"
when proc is_overdue
  print "Bill is overdue"
when proc is_zero
  print "No bill this month"
when proc is_mate
  print "Don't worry too much about this"
endcase
```

See also

when, **otherwise**, **endcase**

circle

Syntax

circle [**fill**] *x*, *y*, ***radius***

Purpose

To draw a circle.

Arguments

x, *y* are the coordinates of the circle centre in OS units.

radius is the radius of the circle in OS units.

Use

This command allows you to draw a circle outline or a filled circle (with the **fill** option). It is only useful in redraw event handlers, since you are not otherwise permitted to draw on the screen.

See also

draw, gcol, move, plot, point, rectangle

clear

Syntax

clear {*var*}

Purpose

To dispose of variables.

Arguments

var is a variable to dispose. If no variables are given, all variables are disposed of.

Use

This command clears variable and procedure definitions. It is very useful where a data structure is created by one handler and must be disposed of by another. You declare and dimension variables in one handler, then use **clear** to get rid of them when they are finished with.

The preferred way of doing this is to declare the variables local to the program or procedure which uses them. However, this may mean that the variables exist longer than they should, which is sometimes important. One reason for early disposal may be that the large number of transient variables dictates that they should only exist where necessary, to save on memory.

The use of **clear** without any parameters is not normally useful within a running program, but it can be useful when trying things out on the command line.

Variables which are 'in use' by the execution stack cannot be cleared. For example, a variable which has been passed to a procedure by reference (using **return**) may not be cleared until the procedure exits. This is common sense, and will not normally present a problem.

See also

dim, local

clg

Syntax

clg

Purpose

To clear the current graphics window.

Use

This command is equivalent to `vdu 16`. It clears the current graphics window to the current graphics background colour.

close

Syntax

(1) **close** ***handle***

(2) **close data** [***filespec***]

(3) **close index** ***filespec*** [, ***indexspec***]

Purpose

(1) To close a file.

(2) To close a database file.

(3) To close one of the indexes of a database file.

Arguments

handle is the handle of the file to close.

filespec is a data file to close. If not given with the **close data** command, all data files are closed.

indexspec is an index to close. If not given in the **close index** command, (or it is −1), all indexes are closed.

Use

The **close** command allows a file to be closed.

The first variant closes a file opened with **@openin**, etc.

The second variant closes a database file and all its index files. Database and index files are automatically opened when required and closed when the application exits. Sometimes, if an application module opens a large number of files it is useful to close them when the module exits, to save memory or to get around a RISC OS-imposed limit on the number of open files.

The third variant is used in similar situations to close index files. This variant is less useful.

Note that data and index files will be reopened automatically when the application next uses them.

Example

```
| close all data files
close data

| close the "pupil" file
close data pupil

| close the "name" index of the pupil file
close index "pupil", "name"
```

See also

@openin, @openout, @openup

create

Syntax

create ***filespec***

Purpose

To create a new record in a file.

Arguments

filespec is the file where the new record is to be created.

Use

This command is used to add a new record to a file. The record is saved to disc from the file's memory buffer, and indexes are updated.

This command will increment the file's record count. The record created is always **@nextfree** ***filespec*** on a stand-alone station.

Files with variable length records are handled as expected. S-Base looks at the value of the 'variable length field' to decide on the size of the record to be saved. If this field is 0 (indicating no items), S-Base will instead use the size of the 'zero length structure' if this is specified.

Example

```
| create a new plant record
plant.name = "New record"
create plant

| create a new transaction
tran.type = TRANT_ORDER
tran.num_items = 3
...
create tran
```

See also

load, save, update, delete

def local error

Syntax

def local error

...

enddef

Purpose

To define a local error handler.

Use

This command allows you to define a piece of code to be executed when an error occurs. The error handler is local to the current procedure/ program/handler, etc. so will only apply within that object.

When S-Base reaches a **def local error** in the program, the error state is automatically set to 1 (see **error set**) and S-Base continues executing after the **enddef**. If an error occurs later in the object, S-Base will set **@errnum** and **@errstr** and return to the first line of the error handler. If the error handler does not exit the object (e.g. leaving a procedure by **return**), then execution will continue right through the error handler and past the **enddef**. Generally an error handler will use **return** to exit after reporting the error to the user.

Note that there are some restrictions on error handlers. For example, imagine an error handler defined in a procedure which opens a card and polls it. An error occurring in the card's click handler would cause S-Base to return to the start of the error handler. The card would have to be disposed of by the error handler since S-Base would not execute the normal **card dispose** command.

Using error handlers in this way (where S-Base must jump from inside a handler) is not recommended. You would normally provide a separate error handler in the card's click handler.

If the error state is changed to other than 1, then the error handler will not be called, since S-Base will process the error either by halting the application, or by setting **@errnum** and **@errstr**.

If an error occurs in an error handler, the previously defined error handler is called with the new error in **@errnum** and **@errstr**. If there are no previous error handlers, the application will terminate with an error. It is common to have a 'global' error handler in the `startup` program.

Example

```
def proc process_file
local fd = -1

| the error handler
def local error
   error @errnum, "The following error has occurred: " +
@errstr
   if fd != -1 then close fd  {close the file if it was
opened}
   return FALSE
enddef

{ process the file. If an error occurs it will be
reported and the procedure will exit }
fd = @openin "file"
proc process (fd)
close fd

| all ok
return TRUE
enddef
```

See also

error set

def proc

Syntax

def proc ***procname*** [({***arg***}$^{+}$)]

...

enddef

Purpose

To define a procedure.

Arguments

procname is the name of the procedure to define.

arg is a formal variable to receive an argument to the procedure.

Use

This command provides S-Base with the definition for a procedure. ***procname*** must not already exist as a variable of any type other than *unknown*.

Any number of arguments may be provided. Any calling statement must

specify values for all the arguments. The syntax for an argument ***arg*** is:

[**return**] ***varname***

The incoming value will be assigned to ***varname*** for the duration of the procedure, whereupon the old value of ***varname*** will be restored. If the **return** construct is used, the calling value must be a variable, and this variable will be assigned with the value of the formal variable on exit from the procedure. This allows procedures to return values through their parameters.

Procedures can also return a single value by means of the **return** statement, embedded at any point in the procedure. When S-Base encounters this statement, it exits the procedure immediately, returning the value given. This allows procedures to act as simple functions, converting parameters to a single return value.

Procedures can be defined within the program, procedure, or handler resources. However, it is likely that you will find it convenient to put most procedures in the procedure resource. It is perfectly acceptable to have more than one procedure per resource, and these procedures are available to any part of the S-Base application.

Procedures can be defined locally within program, handlers, etc. by preceding the definition with a **local** statement. This is a useful way around the restriction that no two procedures can have the same name. For example, if both the order list and within that, the order editor programs contain a procedure called `add_record`, you would have to precede the inner definition in the order editor with `local add_record`.

Example

The following procedure calculates the maximum and average lengths of plant names for a range of records. It would be useful in a report which needs to know exactly how much space will be required for the plant names it is about to print out. It ignores deleted records and returns the number of records counted.

```
def proc calc_plantname_stats (from, too, return max,
return avg)
local i,len,recs=0
max = 0
avg = 0
for i=from to too
  if !@deleted (plant,i)
    recs += 1
    load plant,i
```

```
    len = @len @trim plant.name
    avg += len
    max = @max (max, len)
  endif
next
avg /= recs
return recs
enddef
```

The procedure might be called like this:

```
local max_len, avg_len

rec_count = proc calc_plantname_stats (1, @recordcount
plant, max_len, avg_len)
print "Total records counted: ",rec_count
print "Maximum length: ",max_len
print "Average length: ",avg_len
```

See also

enddef, local, return

delay

Syntax

delay on
delay off
delay ***percentage***

Purpose

To control an hourglass or some other visual means of indicating a delay.

Arguments

percentage is the number to place under the hourglass as an indication of how far the application is through the task.

Use

This command allows applications to signal a delay in user interaction. While the hourglass is on the screen, the user will generally be happy to wait until it disappears, particularly if a percentage indication is displayed also.

To turn the hourglass on use **delay on**. You can then use **delay** ***percentage*** as required with ***percentage*** between 0 and 100. Finally, remove the hourglass with **delay off**. There is always a small delay between the **delay on** command executing and the hourglass appearing. This is so that very short delays will not cause an unnecessary flicker of the hourglass.

Example

Note that in the example below the **delay off** command is always executed whatever happens during the importing. It is good practice to put all the processing code in a separate procedure, since the presence of a **return** statement anywhere would otherwise cause the hourglass to remain on.

```
| import procedure, returns number of records imported
def proc import (return count)
local est_togo
count = 0
{ get the next record along with an estimate of how many
are left }
while proc import_record (est_togo)
   proc save_record  {save the record}
   count += 1
   delay count * 100 / (count + est_togo)  {update the
hourglass}
endwhile
enddef

| import the data
local rec_count
delay on
ok = proc import (rec_count)
delay off
...
```

delete

Syntax

delete ***filespec, record***

Purpose

To delete a record from a file.

Arguments

filespec is the file to access.

record is the record to delete.

Use

This command allows you to mark a record as deleted, removing it from all indexes associated with ***filespec***.

A deleted record can still be loaded using its record number. However, it is illegal to **update** a deleted record (although you may use **save**). Similarly a deleted record may not be deleted again.

Functions such as **@find** and **@loadindex** will never return a deleted record.

A deleted record will have a non-zero `system` field. You can undelete all records in a file by altering their `system` fields to 0, then using **reindex** to rebuild all the indexes.

Example

```
delete title, 34
```

See also

@deleted, pack, create

dim

Syntax

dim {*datatype var* [*subscripts*] }+

Purpose

To 'dimension' variables, set aside space reserved for their use, in effect 'declaring' in advance that they'll be needed.

Arguments

datatype gives the data type for ***var***.

var is a variable to declare.

subscripts is a list of subscripts for the variable each beginning with '`[`' and ending with '`]`', for example '`[3] [4]`'.

Use

The **dim** command declares the type and dimensions of an array. Once a variable has been declared with this command, its type cannot change. Any attempt to assign a value of a different type will result in the value being changed to the variable's type. If this is not possible a *type mismatch* error will result.

The ***datatype*** consists of a single letter, possibly followed by a modifier. The following types are used:

Letter	Type	Comments
d	Date	Stored as four bytes.
f	Float	May be followed by a digit (4 or 8) giving the number of bytes in the value. The default is 8.
i	Integer	May be followed by a digit (1-4) giving the number of bytes in the value. The default is 4.
s	String	Variable length.

t	Time	Stored as four bytes.
x	X-array	Only valid for arrays. Every element of the array can be of a different type.
z	Structure	Must be followed by the file label for the structure. Each element is the size of the file structure. For example `dim z pupil x` defines a variable x containing a pupil record.

If ***subscripts*** is given, it is a list of subscripts for the variable, each beginning with '`[`' and ending with '`]`'. The expression within these square brackets is the value of the subscript – the size of this dimension of the array. Arrays are limited to three dimensions so up to three subscripts may be given for a variable — the values are limited by available memory. It is not valid to put all subscripts within one set of brackets separated by commas.

Where more than one subscript is given, S-Base multiplies them all together to get the total number of elements in the array. Internally, all accesses to arrays are converted to a single integer subscript. For example, imagine a declaration like:

```
dim i a [10] [20]
```

Here an array with 10 * 20 = 200 elements is declared. They are arranged in memory as a[0][0], a[0][1], a[0][2]...a[0][19], a[1][0]...a[19][19]. That is, a[0] can be thought of as an array of 20 integers a[0][0] to a[0][19]. S-Base will convert a subscript [x][y] to a single integer subscript using the formula y + x * 20. For a three-dimensional array:

```
dim i b [10] [20] [30]
```

with 6000 elements, S-Base converts [x][y][z] using the formula z + y * 30 + x * 20 * 30. You should keep this in mind when using functions and commands which operate on arrays. For example, **@findelem** returns a single integer subscript to indicate where its match was located. In the array '`b`' above, a return value of 2500 would mean

```
b [4] [3] [10]
```

since 10 + 3 * **30** + 4 * **20** * **30** = 2500.

The variable must not exist as a valid variable already (although it may have been declared in a previous **local** command). Variables declared with **dim** cannot change type with future assignment. For example, attempting to assign the integer 5 to an element of a string array will fail.

X-arrays have separate type information for each element. However, the types of elements of X-arrays cannot be declared except by assignment. Elements initially start off as type 'Unknown'.

Procedures cannot be declared using **dim**. These must be defined along with their contents, in a **def proc**...**enddef** construct.

Structure variables (type z) can hold the value of a record from a file of the same type. Structure arrays are useful for holding many records from a file for analysis, etc. Fields from the structure variable are referenced using dot (.) in the same way as fields from a file: for a variable *x*, field *y*, use '`x.y`'.

Example

```
{ create an array of 10 integers and assign the squares
as values }
local i, square
dim i square [10]
for i=0 to 9
   square [i] = i * i
next

| set up an array matching strings to numbers
local match, i
dim x match [50] [2]
for i=0 to 49
   match [i] [0] = "The Answer to Life..."
   match [i] [1] = 42
next
```

See also

local, clear

draw

Syntax

draw [by] *x, y*

Purpose

To draw a line.

Arguments

x, y are the OS coordinates of the end of the line.

Use

This command allows you to plot a line from the current graphics cursor position to *x, y*. If **by** is given, the coordinates are relative to the current graphics cursor position, otherwise they are absolute. **draw** and **draw by**

are equivalent to **plot** 1, *x*, *y* and **plot** 5, *x*, *y* respectively. It is only really useful in redraw event handlers, since you are not otherwise permitted to draw on the screen.

If the graphics cursor is not known to be in the correct position for the start of the line, you can use **move** first.

See also

circle, gcol, move, origin, plot, point, rectangle, clg

else

Syntax

if *expr*
...(part 1)
[**else**
...(part 2)]
endif

Purpose

To conditionally execute statements.

Arguments

expr is an expression which determines which part of the **if** statement is executed.

Use

This command marks the part of an **if** statement which will be executed if *expr* evaluates to false (0). See **if** for more details.

See also

then, if, endif

endcase

Syntax

endcase

Purpose

To indicate the end of a **case** statement.

Use

See **case** for details.

See also

case, when, otherwise

enddef

Syntax

def ...

...

enddef

Purpose

To mark the end of a definition.

Use

This statement marks the end of a **def proc** or **def local error** definition.

See also

def proc, def local error

endwhile

Syntax

while ***expr***

...

endwhile

Purpose

To mark the end of a **while** loop.

Use

See **while** for details.

See also

while

error

Syntax

error ***number, message*** [**,** ***flags*** [**,** ***result*** [**,** ***title***]]]

Purpose

To report an error.

Arguments

number is the error number (not usually displayed).

message is the message to display in the error dialogue.

flags is a set of flags for the command (0 by default).

result is an integer variable indicating which icon was clicked on (not returned by default).

title is the text to place in the window title of the error dialogue ("`Error from ...`" by default, where ... is the application name.

Use

This command allows an error to be reported by an application. All tasks will stop executing until the user clicks OK or CANCEL.

The flags are as follows:

Flag	Meaning
1	Display an OK button.
2	Display a CANCEL button.
4	Highlight CANCEL rather than OK as the default choice.

Add the flags together to obtain the required value. The Window Manager will display an OK button if you request neither OK nor CANCEL.

result should be an integer variable to receive the number of the icon clicked on. This will be 0 for nothing selected (not normally returned), 1 for OK and 2 for CANCEL.

Example

See **def local error** for an example.

See also

def local error, @errnum, @errstr

error set

Syntax

error set ***state***

Purpose

To change the current error state.

Arguments

state is the new error state (0, 1 or 2).

Use

error set allows an application to determine how S-Base will handle errors. There are three error states available.

In state 0 S-Base will halt the application and report the error. S-Base will load the offending object and display it for alteration. The application does not get a chance to handle the error.

In state 1 S-Base sets up **@errnum** and **@errstr** and calls the current error handler, which is expected to deal with the error. Other than that, execution continues normally. If there is no current error handler, S-Base

will behave as with state 0.

In state 2 S-Base sets up **@errnum** and **@errstr** but otherwise ignores the error and keeps executing from the next line. The application is expected to check the value of **@errnum** after each line to make sure no error has occurred. If an error has occurred, S-Base will set **@errnum** to a non-zero value.

A **def local error** command automatically sets the error state to 1. Note that setting the error state to anything other than 1 will effectively remove the error handler temporarily, until the error state is set back to 1.

Example

```
| load a record with an error state of 2
error set 2
load video, 5
error set 1

{ if 'load' caused an error, handle it, else process the
record }
if @errnum
   proc report_error
else
   proc process_record
endif
```

See also

def local error, **@errnum**, **@errstr**

exit

Syntax

exit [*return_code*]

Purpose

To exit an application abnormally.

Arguments

return_code is the completion code to return to the calling program (0 by default).

Use

This command allows an application to exit at any time, perhaps because of a fatal error which cannot be corrected, or because the user has asked to quit the application.

S-Base terminates the application normally, closing all open files, etc.

exit allows an application running outside the S-Base environment to

provide a ***return_code*** for the calling application. This is usually 0 for success, or some other value (generally 1 to 255) if an error occurred. Within the S-Base environment, this value is not useful.

This command should not be used at the end of the `startup` program. When S-Base finishes executing this program it will exit anyway.

file set field

Syntax

file set field ***filespec,fieldspec*** [***subscripts***], [***operator***] ***value***

Purpose

To assign a value to a field.

Arguments

filespec is the file containing the field (either integer, structure or string).

fieldspec is the field to assign (either integer or string).

subscript is the field's subscript giving the element of the field to use.

operator is the assignment operator to use for the assignment (equals (=) by default).

value is the value to assign.

Use

This command allows a value to be assigned to a field as with a normal `file.field = ...` assignment. The difference is that you can specify the file and field using integers or even strings. This is useful when you want a piece of code to do the same thing with two different fields of a file, or even with two different files. You can pass in (as a string or integer) the file and field to use in that piece of code.

Often you can do this other ways. For example, you could write a procedure which accepts parameters passed by reference. This command should be used sparingly as it reduces the readability of the code.

Example

You are left to decide for yourself when this command should be used, so no full-context example is given.

Add 9 to `title.price[2]`:

```
file set field "title", "price"[2], += 9
```

Set `title.title` to `"Star Wars"`:

```
file set field "title", "title", "Star Wars"
```

See also

@filefield

flush

Syntax

(1) **flush**

(2) **flush** ***handle***

Purpose

(1) To flush all data file and index buffers.

(2) To flush a file's buffer.

Arguments

handle is the handle of the file to flush.

Use (1)

This command forces S-Base to save to he relevant files all data held in memory. After this, should the machine go down without getting a chance to exit normally, the data and indexes should still be intact.

S-Base performs this operation automatically when the machine is idle.

Use (2)

This command allows you to flush a single file (opened with **@openup** for example) to the filing system. All cached data will be written, as if the file had been closed.

Example

In this example we use flush after each file alteration to prevent loss due to machine failure, for example.

```
def proc edit_file
local fd, pos, data, done

| open the file
fd = @openup "userdata"

| allow the user to alter the data file
repeat
   | get the changes
   done = proc get_edits (pos, data)
   if !done
      | write them to the file
      proc write_edits (fd, pos, data)

      | flush the file
```

```
      flush fd
   endif
until done

| close the file
close fd
```

See also

archive, close, @openup, @openout

for

Syntax

for ***var*** **=** ***start*** **to** ***end*** [**step** ***step***]

...

next

Purpose

To execute a controlled loop (one or more times).

Arguments

var is the control variable to use in the loop. This must be an integer variable.

start is the initial value of the control variable.

end is the terminating value of the control variable.

If ***step*** is specified, it is the value which is added to the control variable at the end of each iteration of the loop. The default is 1.

Use

This command is used to execute a sequence of statements between **for** and **next** repeatedly until the control variable exceeds its termination value.

The termination test is done at the **next** statement, after ***step*** has been added to the control variable. Thus the statements within the construct are always executed at least once. This is not the case with **ffor** (see later).

Example

```
| print out first 50 plant names
for i=1 to 50
  load plant,i
  print plant.name
next
```

```
| total up sales
total = 0
for i=0 to NUM_DEPARTMENTS-1
  total += dept [i].sales
next
```

See also

ffor

ffor

Syntax

ffor *var* = *start* **to** *end* [**step** *step*]

...

next

Purpose

To execute a controlled loop.

Arguments

var is the control variable to use in the loop. This must be an integer variable.

start is the initial value of the control variable.

end is the terminating value of the control variable.

If *step* is specified, it is the value which is added to the control variable at the end of each iteration of the loop. The default is 1.

Use

This command is used to execute a sequence of statements between **ffor** and **next** repeatedly while the control variable has not exceeded its termination value.

The termination test is done at the **ffor** statement before the statements within the loop are executed. Thus the statements within the construct will not be executed at all if the termination test fails at the start of the loop.

Example

```
| print out all plant names
ffor i=1 to @recordcount plant
  load plant,i
  print plant.name
next
```

See also

for

gcol

Syntax

(1) **gcol** ***colour***

(2) **gcol** ***action, colour***

Purpose

To set the current graphics colour.

Arguments

colour is the graphics colour to use for future plotting. If ***colour*** < 128, the foreground colour is set, otherwise the background colour is set to (***colour*** −128).

action is the graphics plot action (0 by default).

Use

This command allows you to set the colour of lines and text plotted on the screen. It is only useful in redraw event handlers since drawing on the screen is not otherwise permitted. You cannot assume any particular graphics colour is selected at the start of a redraw handler.

The value of ***action*** can be one of the following (this information is repeated from the RISC OS Programmers Reference Manual):

Value	Meaning
0	Overwrite colour on screen with ***colour***
1	OR colour on screen with ***colour***
2	AND colour on screen with ***colour***
3	EXCLUSIVE OR colour on screen with ***colour***
4	Invert colour on screen (ignore ***colour***)
5	Leave screen unchanged (no action)
6	AND colour on screen with (~ ***colour***)
7	OR colour on screen with (~ ***colour***)

See the RISC OS Programmers Reference Manual for the other action codes which are available. In a Wimp environment, action 0 is nearly always used.

The colours referred to by ***colour*** depend on the current palette. Look at the Palette icon on the icon bar: the top left colour is 0 and the bottom right colour is 15. You should not normally change this palette as it is under

user control.

See also

circle, clg, draw, move, origin, plot, point, rectangle

if

Syntax

(1) **if** ***expr*** **then** ***command***

(2) **if** ***expr***
...(part 1)
[**else**
...(part 2)]
endif

Purpose

To conditionally execute statements.

Arguments

expr is an expression which determines which parts of the **if** statement are executed.

Use

This command allows a sequence of commands to be executed on the basis of whether an expression is true. The first syntax, the one-line version, will execute ***command*** only if ***expr*** evaluates to true (anything other than 0). If ***expr*** evaluates to 0, S-Base will skip to the next line without executing ***command***.

The second syntax, the multi-line version, will if ***expr*** is true (other than 0), execute part 1 and skip part 2. However, if ***expr*** is false (0), part 1 will be skipped and part 2 executed. Part 2 need not be present, in which case it becomes a simple **if** statement with no **else** clause.

Note that the difference between the two syntaxes is the presence or absence of the **then** keyword. If this is not present, the statement is read as a multi-line **if**, and S-Base will expect an **endif** to follow. Otherwise, the statement is read as a single-line **if**, and S-Base will not expect a following **endif**.

Example

```
if plant.price > 10.00
  print "This is an expensive plant"
else
  print "This plant isn't too expensive"
endif
```

```
if plant.row = 0 then print "Row is invalid"
```

See also

then, else, endif

import

Syntax

import [**text**] ***handle***, {***filefield*** | ***var*** [**ext** ***size***] }+

Purpose

To import binary or text values from a file.

Arguments

handle is the handle of the file to import from (opened with **@openin**, etc.).

filefield is a file field to place a value into.

var is a variable to place a value into.

size is only available if ***filefield*** is an external file field. It gives the number of bytes to read from the file (the rest of the file by default).

Use

This command is useful when you wish to read values from a non-database file. You may wish to import data from another type of database, for example. If **text** is specified, S-Base reads a line from the file of text from the file and converts it to a value. Otherwise, S-Base reads raw binary data from the file.

S-Base determines the data type by looking at the file field or variable you give in the arguments. For example, if you provide a string variable as ***var***, S-Base will read a string from the file and place it in ***var***. Similarly, if you provide an integer file field as ***filefield***, S-Base will read an integer from the file and place it in ***filefield***. Variables which do not have a type yet (e.g. just declared locally but not assigned) are assumed to be integers. No type information is stored in or retrieved from the file.

For external file fields, the number of bytes to read into the file field can be specified after **ext**. S-Base will destroy the old contents of the file field and store ***size*** bytes from the file instead.

See **archive** for a description of the binary format used by S-Base.

Example

Here a foreign database file is opened for input to find out how many records it contains (stored in 2 bytes at offset 256 in the file):

```
def proc get_record_count (fname)
| open the file
local fd = @openin fname, rec_count

| read in the number of records in the file
set ptr fd, 256
import f, rec_count

| convert to a 16 bit value
rec_count &= &ffff

| close the file & return
close fd
return rec_count
enddef
```

See also

archive, set ptr, @openin

load

Syntax

load ***filespec*** **,** ***recnum*** **[,** ***real_filespec*** **,** ***bytecount*** **]**

Purpose

To load a record from a file into its record buffer.

Arguments

filespec is the file from which the record is to be loaded.

recnum is the record to load.

real_filespec is the file whose buffer is to be loaded with the data. If not specified, it is taken to be the same as ***filespec***.

bytecount is the number of bytes to load. If not specified (so that ***real_filespec*** is not given either), it is taken to be **@recsize** ***filespec***.

Use

This command is used to load data from a file. For fixed-length record files, the optional part of the command is generally left out. In this case the whole record is loaded into the file's record buffer.

For variable-length records, S-Base would normally load the entire record into memory. This is possible because S-Base can look at the record length on disc (the distance between the start of this record and the next). However, sometimes it is desirable for the application to specify the record length. For example, if in a file with variable length records, each record is

in fact of a different type, the application can load the first few bytes of the record to determine the record type, then load the rest as it wishes.

Obviously, unless the entire record is loaded, not all fields will have the correct value. Loading half the record will only update half the fields – the other half will keep their original values. Care should be taken that data is loaded before it is used.

Example

```
| load in record 50 & print the plant name
load plant,50
print "The name of record 50 is ",plant.name

{ figure out the type of a record, and then load it into
the correct memory buffer. 'tran.type' is the first
field of the 'tran' file, and occupies 4 bytes }
load tran,5,tran,4  {load 4 bytes into 'tran' record
buffer}
case tran.type
when 1
   load tran,5,invoice,@recsize invoice
   print "Total invoice value ", invoice.value:1:2
when 2
   load tran,5,order,@recsize order
   print "Total order value ", order.value:1:2
otherwise
   proc err ("Unknown type "+@str tran.type)
endcase
```

This example loads a whole transaction into the `tran` file buffer. S-Base will give an error if the record is too large for the record buffer. Note that S-Base ignores the file's 'zero-length structure' and 'variable length field' when loading records from disc.

```
load tran,5
```

See also

create, save, update

local

Syntax

local {*var* [= *expr*] }+

Purpose

To declare variables local to the construct containing them.

Arguments

var is the variable to make local.

expr is an expression to assign to the variable.

Use

local stores the value and state of the variables listed on the execution stack. Following this, the variables may be used within the construct containing the **local** command without interfering with variables of the same name outside the construct. If ***expr*** is given, it is assigned to the variable after its original value has been saved.

local is useful within a procedure where the use of variables local to that procedure should not affect variables outside it. It is also useful for making procedures defined in a program local to that program.

Variables are generally made local to procedures, handlers and programs. However, it is legal to make variables local to most constructs: **while...endwhile**, **for...next**, etc.

Variables and procedures can be made local before being declared. This is necessary for local arrays: **local** should precede **dim** for a variable. A local variable with no type yet allocated (by assignment or declaration) will have the type 'Unknown'.

For a local procedure, precede the procedure definition (**def proc**) with a line containing a **local** statement for the procedure's name. Although the procedure name will appear in the variable list as soon as it is declared local, you obviously cannot call the procedure until you have defined it with **def proc**.

Variables which did not exist before being encountered in a **local** statement will be destroyed on exit from the construct. Variables which did exist will have their original value restored.

Example

```
local test

def proc test(x)
local i
for i = 1 to x
   print i
next
print "Within procedure, x is ",x
```

```
enddef

x = 5
print "x is ",x
proc test (12)
print "After procedure, x is still ",x
```

See also

dim, def proc

lvar

Syntax

lvar

Purpose

To provide a list of all variables in use.

Use

This command is generally not used in programs but can be useful for debugging from the S-Base command line. It generates a list of all variables currently in use, including procedures.

Example

```
a = 1
b = 2
c = 3
lvar
```

menu clear protect

Syntax

menu clear protect ***handle*** {, ***entry***}

Purpose

To unfade menu entries.

Arguments

handle is the handle of the menu to alter.

entry is an entry to unfade. This can be either the entry number or the text within it, in which case the first item matching the text will be used.

Use

This command is useful for unfading entries after they have been created. It is generally used only in menu maker handlers.

With no ***entry*** parameters, **menu clear protect** unfades all entries. This form of the command is often used before some entries are faded (using **menu set protect**).

Example

See **menu new** for an example.

See also

menu new, menu set protect

menu clear select

Syntax

menu clear select *handle* {, *entry*}

Purpose

To untick menu entries.

Arguments

handle is the handle of the menu to alter.

entry is an entry to untick. This can be either the entry number or the text within it, in which case the first item matching the text will be used.

Use

This command is useful for unticking entries after they have been created. It is generally used only in menu maker handlers.

With no ***entry*** parameters, **menu clear select** unticks all entries. This form of the command is often used before some entries are ticked (using **menu set select**).

Example

See **menu new** for an example.

See also

menu new, menu set select

menu dispose

Syntax

menu dispose *handle*

Purpose

To dispose of a menu which is out of date or no longer required.

Arguments

handle is the handle of the menu to dispose.

Use

This command is useful when the menu must be rebuilt because the situation has altered. For example, a menu which shows a list of titles for selection must be updated when a new title is added, or an existing one deleted.

menu dispose removes the menu from memory. Subsequent calls to

@menufind with the menu's label will recreate the menu using the definition in the menu resource.

Example

See **menu new** for an example.

See also

menu new

menu insert

Syntax

menu insert ***handle, entries***

Purpose

To add entries to a menu.

Arguments

handle is the handle of the menu.

entries specifies the entries to add to the menu.

Use

This command allows you to insert entries into a menu once you have created it. It adds menu entries after any existing ones.

The ***entries*** part is a string containing a list of entries each separated by a comma (,) or a bar (|). Entries separated by a comma will follow each other immediately; entries separated by a bar will have a line between them.

You can precede each entry with an option character, one of the following:

!	Put a tick to the left of this menu entry.
~	Fade this menu entry so the user cannot select it.
>	Put an arrow to the right of this entry. The user can move the pointer over the arrow to open a dialogue.

Within the string, entry text must be enclosed in double quotes (i.e. ") if the text to be displayed contains any of the above characters (or comma or bar). If ***entries*** is given as a literal string rather than a variable, follow the normal rules – to insert a double quote in a literal string you must use two double quotes, e.g. `menu insert m, """Funny !~ characters!"""` is used to insert the string `"Funny !~ characters!"` (the quotes will appear in the menu item).

Entries may include a keyboard short-cut by adding a colon (:) followed by the keyboard short-cut (e.g. "Delete:^X"). Note that this keyboard short-

cut is not automatically implemented by S-Base. You must handle the keypress events in your application and perform the operation yourself.

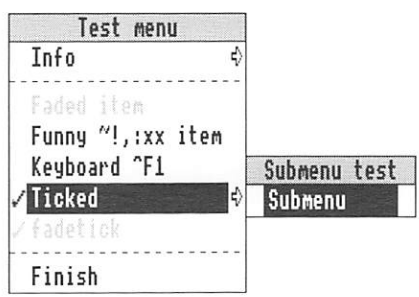
Note that S-Base assumes a comma between entries defined by separate **menu insert** commands. If you want a bar instead (so that entries are separated by a line) you must start the ***entries*** parameter of the second **menu insert** command with a bar (|). For example, to get a bar between the two entries 'one', 'two' and 'three' below, you must insert the bar as shown:

```
menu insert m, "one|two"
menu insert m, "|three"
```

menu insert does not allow entries to be inserted anywhere but at the end of the menu.

Example

```
| An icon bar menu
menu new m,"s-base iconbar","S-Base"
menu insert m,">Info|>Configure,>Open window"
menu insert m,">Close window,Remove contexts"
menu insert m,"New application,~Network directory,Quit"
```




```
| A test menu
```

```
menu new m,"test","Test menu"
menu insert m,">Info|~Faded item,""Funny ~!,:xx item"""
menu insert m, "Keyboard:^F1,!Ticked,!~fadetick|Finish"

| create the submenu
menu new mm,"test2", "Submenu test"
menu insert mm, "Submenu"

| attach the submenu
menu set menu m,4,mm
```

See also

menu new

menu new

Syntax

menu new ***handle, label, title***

Purpose

To create a menu.

Arguments

handle is an integer variable into which the handle for the new menu will be placed.

label is the menu's label. This is used to identify the menu by name.

title is the menu's title, displayed above the menu entries.

Use

This command is used to create a new menu. Menus are generally used to perform actions in an application. For example, most applications have a menu accessible from their icon bar icon.

The menu is identified by its label. You can use this label to recall the menu later with **@menufind**.

You should only recreate a menu when you know it has changed, and you should always dispose of the old menu (using **menu dispose**). Otherwise, you might create a number of different menus all with the same label, and be unable to determine which is the correct one. **menu new** should normally only be used in a menu resource, and ***label*** must match the label of the menu resource so that **@menufind** will find it.

This command is mainly used in *menu maker* handlers, if the menu required does not yet exist.

Example

This example is a typical menu resource (called `browser`):

```
local m

| create the menu
menu new m,"browser","Browser"
menu insert m,"Next record,Previous record"
menu insert m,">Record"
menu insert m,">Goto record,Show address|Quit"
menu set m
```

This example is a typical menu maker handler:

```
| get the menu handle
m = @menufind "browser"

| clear any previous ticks & fades
menu clear select m
menu clear protect m

{ tick the 'show address' field if this is the current
setting }
if show_address then menu set tick m,4

| fade 'previous' if we are at the first record
if @record curfile = 1 then menu set protect m,1

| fade 'next' if we are at the last record
if @record curfile = @recordcount curfile
   menu set protect m,0
endif

| fade 'goto' if there is only one record
if @recordcount curfile = 1 then menu set protect m,3

| change item 2 (the current record) as required
menu set m, 2, "Record '" + @str (@record curfile) + "'"

| return this menu
menu set m
```

See also

@menufind, @menuselect

menu set

Syntax

(1) **menu set** ***handle***

(2) **menu set** ***handle***, ***entry***, ***string***

Purpose

(1) To display a menu on the screen.

(2) To alter an entry of a menu.

Arguments

handle is the handle of the menu to display/alter.

entry is the entry to alter (numbered from 0).

string is the new string to put in the entry.

Use (1)

This command is generally used at the end of a menu maker handler to tell S-Base which menu to display on exit from the menu maker. The menu must be ready for use. It is also used for the same purpose at the end of a menu resource.

If you add entries to a menu after you have used this command they may not appear.

You can also use this command in click handlers to the same effect. This is useful where you have a pop-up icon. When the user clicks on it you can tell S-Base to open a menu using **menu set**. The menu select handler is called as usual when the user selects a menu option.

Use (2)

This command allows you to alter an entry of an existing menu. Note that only the text in the entry changes. You cannot start ***string*** with, for example, an exclamation mark (!) to tick the item. Use **menu set select** and related commands for this. If the entry is ticked it will stay ticked.

This command is important for two reasons. Firstly, it is slow to recreate a menu every time an entry changes. Secondly, recreating a menu will move it in memory, so the Window Manager will not realise it is the same menu. When the user clicks on a menu with the adjust button, the menu will not come back.

A menu should only be recreated if its structure changes. However, changing the structure of a menu is not recommended as it confuses the user (you should fade unavailable entries instead). So recreating a menu is a fairly rare event — perhaps because the user has swapped between short menus and full menus. In this case, the menu has changed and it is ok for it

to move. Clicking adjust on the menu cannot cause the menu to remain.

Example

See **menu new** for an example.

See also

menu new

menu set menu

Syntax

menu set menu ***handle***, ***entry***, ***sub_handle***

Purpose

To attach a submenu to a menu.

Arguments

handle is the handle of the main menu.

entry is the entry number (from 0) of the main menu to which the submenu will be attached.

sub_handle is the handle of the submenu.

Use

menu set menu allows hierarchical menus to be created by attaching submenus to main menus.

Once the submenu is attached it must not be altered with **menu insert**. If the submenu is altered it must be attached to the main menu again. If the submenu is disposed of or recreated while attached to the main menu, then the main menu cannot be used until the new submenu is attached or the main menu itself is recreated.

Example

See **menu new** for an example.

See also

menu new

menu set protect

Syntax

menu set protect ***handle*** {, ***entry***}

Purpose

To fade (protect from selection) menu entries.

Arguments

handle is the handle of the menu to alter.

entry is an entry to fade. This can be either the entry number or the text within it, in which case the first item matching the text will be used.

Use

This command is useful for fading entries after they have been created. It is mostly used in menu maker handlers for fading unavailable choices.

With no ***entry*** parameters, **menu set protect** fades all entries. You could then unfade some of them as required (using **menu clear protect**).

Example

See **menu new** for an example.

See also

menu new, **menu clear protect**

menu set select

Syntax

menu set select ***handle*** {, ***entry***}

Purpose

To tick menu entries.

Arguments

handle is the handle of the menu to alter.

entry is an entry to tick. This can be either the entry number or the text within it, in which case the first item matching the text will be used.

Use

This command is useful for ticking entries after they have been created. It is mostly used in menu maker handlers for ticking the currently selected entry.

With no ***entry*** parameters, **menu set select** ticks all entries. You could then untick some of them as required (using **menu clear select**).

Example

See **menu new** for an example.

See also

menu new, **menu clear select**

move

Syntax

move [**by**] *x*, *y*

Purpose

To move the graphics cursor.

Arguments

x, *y* are the OS coordinates to move to.

Use

This command allows you to move the current graphics cursor position to *x*, *y*. If **by** is given, the coordinates are relative to the current graphics cursor position, otherwise they are absolute. It is equivalent to **plot** 0 or 4, *x*, *y*. It is only useful in redraw event handlers, since you are not otherwise permitted to draw on the screen.

See also

circle, **draw**, **gcol**, **origin**, **plot**, **point**, **rectangle**, **clg**

next

Syntax

for *var* = *start* **to** *end* [**step** *step*]
...
next

Purpose

To mark the end of a **for** or **ffor** loop.

Use

next should be used at the end of a **for** or **ffor** loop. Note that you *cannot* specify the loop's control variable.

See **for** and **ffor** for the use of this command.

See also

for, **ffor**

origin

Syntax

origin *x*, *y*

Purpose

To set the graphics origin.

Arguments

x, *y* are the OS coordinates of the new origin.

Use

This command allows you to change the graphics origin. All graphics coordinates are relative to this origin, which is generally 0,0. It is only really useful in redraw event handlers, since you are not otherwise permitted to draw on the screen.

See also

circle, **draw**, **gcol**, **move**, **plot**, **point**, **rectangle**, **clg**

oscli

Syntax

oscli ***string***

Purpose

To pass a command to RISC OS.

Arguments

string is a string containing the command to be executed.

Use

This command allows RISC OS *star commands* to be executed from within S-Base. The effect is as if the user had typed the contents of ***string*** at the command line. The string does not need to start with a star.

You should be careful not to do anything which might cause S-Base problems. For example, if `*shut` is executed, S-Base will be unable to access any of its files. It is good programming practice to put **oscli** commands within a separate procedure so any later problems caused by them can be located quickly.

Example

```
| a procedure to remove a file
def proc remove (filename)
   oscli "remove "+filename
enddef
```

See also

sys

otherwise

Syntax

otherwise

Purpose

To mark the default clause in a **case** statement. Statements after **otherwise** will be executed only if the expression after the **case** statement matches none of the expressions after the **when** statements.

Use

See **case** for details.

See also

case, when, endcase

pack

Syntax

pack *filespec*

Purpose

To pack a file down by removing deleted records.

Arguments

filespec is the file to pack.

Use

Deleted records are not actually deleted when you issue the **delete** command. This would be too inefficient, since S-Base would have to copy records around to fill up the space. Every now and then, perhaps automatically, you can use **pack** to remove unwanted records.

This command is useful in a number of situations. You could use **pack** on an address book file, for example, to clear out deleted records. Or you could pack a temporary transaction file after removing some processed records from it. S-Base will shrink the size of the file if necessary.

pack is fairly slow since it involves bulk copying of records. It does not rebuild indexes – use **reindex** *filespec* for this.

You should take care that you do not pack files which are referenced by another file. For example, if in a transaction file you store the record number of a video title which has been hired out to a customer, you cannot pack the video title file. This would change the record numbers of titles, thus rendering the transaction file useless.

See also

delete, reindex

Example

```
| pack the video file
pack video
```

plot

Syntax

plot *code*, *x*, *y*

Purpose

To plot a point, line, circle, rectangle, triangle, etc.

Arguments

code specifies what type of plot to perform.

x, *y* are the OS coordinates to pass to the plot.

Use

This command gives you direct access to the plot drivers. It is only really useful in redraw event handlers, since you are not otherwise permitted to draw on the screen.

You should refer to the RISC OS Programmers Reference Manual for details about plot codes, and the VDU drivers in general.

Example

```
| draw a line from 0,0 to 100,100
plot 4, 0, 0
plot 5, 100, 100
```

See also

clg, circle, draw, gcol, move, origin, point, rectangle

pmenu dispose

Syntax

pmenu dispose [update] ***handle***

Purpose

To dispose of a program menu.

Arguments

handle is the handle of the program menu to dispose of.

Use

pmenu dispose should be used to dispose of a program menu when you have finished with it. This will release memory for other purposes.

pmenu dispose first does a **pmenu hide** then removes the program menu from memory.

If **update** is given in the command, the template used to create the window is updated with the card's position when last hidden. This means that the card will always pop up where the user last left it rather than returning to its original position every time it is recreated (unless the program menu is centred).

Example

See **pmenu new** for an example.

See also

pmenu new

pmenu hide

Syntax

pmenu hide ***handle***

Purpose

To remove a program menu from the screen.

Arguments

handle is the handle of the program menu to hide.

Use

pmenu hide removes a program menu from view. It is useful once the user has selected an option, since the program menu should not normally hang around while the program option is running.

This command does not dispose of the program menu — you must use **pmenu dispose** for this. It simply closes and deletes the window containing it.

Example

See **pmenu new** for an example.

See also

pmenu new

pmenu insert

Syntax

pmenu insert ***handle***, ***sprite***, ***key***, ***text*** [, ***flags***]

Purpose

To add a new item to a program menu.

Arguments

handle is the handle of the program menu to add an item to.

sprite is the sprite to select for this item (use an empty string for none).

key is the key to press for this item.

text is the descriptive text for this item.

flags is a set of options for the item.

Use

pmenu insert creates a new item in a program menu. It should be used after **pmenu new** but before the menu is displayed.

sprite must be the label of a sprite in the application's sprite resource.

The flags are as follows:

Flag	Meaning
1	Insert a space after this item
2	This item is not present

4	This item cannot be selected

Add the flags together to get the required ***flags*** value.

An item marked as not present (flag 2) is still counted as an item, but will not be displayed in the program menu. This allows you to remove items from the pmenu (perhaps because they are temporarily unavailable) without altering the numbering of items. It is more common to use flag 4, since the user may be confused by options appearing and disappearing.

There is no limit to the number of items which can be added to a program menu, although large menus may not fit on the screen. Generally 5-7 items is best.

If new items are inserted into a program menu while it is displayed, it will not be updated. You must use **pmenu hide** followed by **pmenu show** to recreate the menu.

Example

See **pmenu new** for an example.

See also

pmenu new

pmenu new

Syntax

pmenu new ***handle***, ***title*** [, ***columns*** [, ***flags***]]

Purpose

To create a new program menu.

Arguments

handle is an integer variable into which the program menu's handle will be placed.

title is the title of the program menu.

columns is the number of columns in the program menu (1 by default).

flags is a set of options for the program menu (0 by default).

Use

Program menus are useful for allowing the user to navigate around an application.

Large menus arranged in a large hierarchical fashion should be avoided. Use program menus only for moving between major program features.

pmenu new creates a new empty program menu. You can add items to the menu using **pmenu insert**.

A program menu item has three parts: sprite, key and text. *Sprite* is a picture representing the program option, *key* is the key to press to select it and *text* is a short description of the option. At least one of these parts must be present.

The flags are as follows:

Flag	Meaning
4	Invert the item under the mouse pointer as the user moves the pointer
8	Do not display sprites
16	Do not display keys or allow them to select an item
32	Do not display text
64	Centre the pmenu on the screen when opened

Add the various flags together to get the number you require. You will understand the options better if you see **pmenu insert**.

The items of a program menu are numbered from 0, and do not depend on the *key* value.

Program menus are intended to be a simple way of offering a user a set of choices. They should not be used to create huge hierarchical menus in which the user can get lost. If you require more flexibility than provided by program menus, you should use a card with sprite and prompt elements.

Example

```
{ create a main menu with various options. The user can
select an option by clicking on it, or pressing a key 0
to 9 }
pmenu new p,"Main Menu",2,16
pmenu insert p,"add","1","Add record"
pmenu insert p,"edit","2","Edit record"
pmenu insert p,"print","3","Print record"
pmenu insert p,"list","4","Record list"
pmenu insert p,"info","5","File information"
pmenu insert p,"import","6","Import data"
pmenu insert p,"reindex","7","Reindex file"
pmenu insert p,"info","8","Contact information"
pmenu insert p,"print","9","Print pupil cards",1
pmenu insert p,"quit","0","Quit"
```

```
| user has not finished with the menu yet
done = 0

| loop around until finished
repeat
   | display the menu and get the required option
   pmenu show p
   pmenu poll p,option

   | hide the menu and perform the correct option
   pmenu hide p
   case option
   when 0   { 1st item selected - add record }
      proc add_record
   when 1
      proc edit_record
   when 2
      proc pupil_cards
...
   when 9
      done = 1
   endcase
until done

| dispose of the menu
pmenu dispose p
```

See also

pmenu dispose, pmenu insert

pmenu poll

Syntax

pmenu poll ***handle, option***

Purpose

To allow the user to select a program menu option.

Arguments

handle is the handle of the program menu.

option is an integer variable which will be set to the number of the chosen program menu entry (numbered from 0).

Use

pmenu poll should be used after a program menu has been created and displayed on the screen. It will wait until the user selects an option and then set ***option*** to the entry number of that option.

option will be in the range 0 to the number of items in the program menu less one, with 0 being the first item. Note that the *key* part of an item has no relationship to the item number.

Example

See **pmenu new** for an example.

See also

pmenu new, pmenu show

pmenu show

Syntax

pmenu show ***handle***

Purpose

To display a program menu.

Arguments

handle is the handle of the program menu to display.

Use

pmenu show is used to display a program menu on the screen ready for user input. Note that the menu will not appear until the Window Manager is next polled.

A window is created to hold the program menu, and icons are placed in the window. S-Base works out the dimensions and layout automatically.

You should follow this command with **pmenu poll** to accept user input. You should not add items to the program menu once you have issued this command (use **pmenu hide** first).

Example

See **pmenu new** for an example.

See also

pmenu hide, pmenu new

point

Syntax

point [**by**] *x*, *y*

Purpose

To plot a point.

Arguments

x, *y* are the OS coordinates of the point.

Use

This command allows you to plot a point (a single pixel), equivalent to **plot** 65 or 69, *x*, *y*. If **by** is given, the coordinates are relative to the current graphics cursor position, otherwise they are absolute. This command is only useful in redraw event handlers, since you are not otherwise permitted to draw on the screen.

See also

circle, clg, draw, gcol, move, origin, plot, rectangle

print

Syntax

print { **to** *handle* | [*expr*][*fieldspec*] } [;]

Purpose

To convert an expression to a string and display it in the command line window or write it to a file.

Arguments

handle is the handle of a file to print to. If not given (or is −1), output will be directed to the command line.

expr specifies an expression to be converted to a string and displayed (an empty string by default).

fieldspec specifies how the string conversion will take place.

Use

This command complements **@str** as a method for displaying data or sending it to a file in text form.

A ***fieldspec*** consists of up to three parts, all of which are optional. The full syntax is:

[***:/ subtype***] [**:** ***width*** [**:** ***decplaces*** | ***maxwidth***]]

The ***subtype*** part is recognised by a forward slash (*/*) which is followed by a single character indicating the subtype. Legal subtypes are:

Subtype	Meaning
a	Converts to a character in the range 'A' to 'Z'. For example, 0 will convert to 'A', 1 to 'B', and 25 to 'Z'. Other values convert to a space.
c	Converts an integer to its ASCII equivalent. For

	example, 65 converts to 'A', 33 to '!'. Values outside the range 32...255 (and 127) convert to a space.
i	Converts a time to 'interval' format, giving a time interval in years, months, weeks, days, hours, minutes and seconds as '1y2m3w4d5h6u7s'. Only useful parts are included. For example, if the interval is more than a year then hours, minutes and seconds will not be included.
n	Converts a time to 'natural date' format: 'dd/mm/yy'.
m	Converts a date to 'date' format with the month written out, e.g. '02–Jan–92'
t	Converts a time to 'time' format: 'dd/mm/yy,hh:mm:ss'

The ***width*** specifier, if present, is used to pad the converted string out to a certain number of characters. If ***width*** is positive, the string is padded to the left with spaces. If ***width*** is negative, the string is padded to the right. If |***width***| (i.e. **@abs** ***width***) is less than the length of the string, it has no effect.

The ***decplaces*** or ***maxwidth*** field controls the number of decimal places for numeric conversion or the maximum string length for other conversions.

If ***decplaces*** is specified for a numeric conversion, the number is first converted to a float, then the specified number of decimal places are added.

If ***maxwidth*** is specified for a non-numeric conversion (this includes times and dates), the string is truncated to ***maxwidth*** characters after the conversion.

For time types, the default format is '/t', and for date types it is '/n'.

Example

This example prints out a plant list to a file called 'plantlist'. The record number is padded out to 4 characters (right justified), followed by two spaces, followed by the first 20 characters of the plant name:

```
| print out a formatted plant list
local f=@openout "<Plant$Dir>.plantlist"
```

```
ffor i = 1 to @recordcount plant
   load plant, i
   print to f, i:4, :2, plant.name:1:20
next
close f
```

This example shows use of the maximum width field:

```
str = "Hello"
ffor i = 1 to @len str
  print str:1:i
next
```

which gives the output:

```
H
He
Hel
Hell
Hello
```

```
| a test
s:print 1.2345:10:5
   1.23450
s:print 12:10:1,:2,'B'
      12.0  66
s:print 12:/a:5,:2,'B':/c,",",@time:/n:10,"."
    M  B,  29/03/92.
```

See also

@str

proc

Syntax

proc ***procname*** [({***arg***}$^{+}$)]

Purpose

To call a procedure.

Arguments

procname is the procedure to call.

arg is an argument to the procedure.

Use

This command is used to call a procedure. The procedure must previously have been defined by the **def proc** command.

The argument list following the procedure name must match the argument list in the definition. An argument must either be an expression (for normal parameters) or just a variable name (for reference parameters, preceded by **return** in the definition).

If the procedure has no arguments, then the brackets must be omitted. See **def proc** for a full description of the procedure definition.

Example

```
proc add_new_record
a = proc sum_squares (1, 10)
ok = proc get_username (name)
```

See also

def proc

rectangle

Syntax

(1) **rectangle** ***left, bottom, width*** [**,** ***height***]

(2) **rectangle fill** ***left, bottom, width*** [**,** ***height***]

(3) **rectangle** ***left, bottom, width*** [**,** ***height***] **to** ***x, y***

(4) **rectangle fill** ***left, bottom, width*** [**,** ***height***] **to** ***x, y***

Purpose

(1) To plot an outline of a rectangle on the screen.

(2) To plot a filled rectangle on the screen.

(3) To copy an area of the screen to another area

(4) To move an area of the screen to another area, setting the source rectangle to the current background colour.

Arguments

left, bottom are the OS coordinates specifying the bottom left corner of the rectangle.

width is the width of the rectangle in OS units.

height is the height of the rectangle in OS units (by default this is the same as ***width***).

x, y are the OS coordinates specifying the destination rectangle (where the lower left coordinate of the source rectangle will move to).

Use

This command allows you to draw rectangles and optionally fill them. It also allows a rectangle on the screen to be moved or copied to a new location.

rectangle uses the normal plot codes, and all the above effects can be achieved using the standard **plot** command. It is generally only useful in a redraw event handler.

See also

circle, clg, draw, gcol, move, plot, point

reindex

Syntax

reindex *filespec* [, *indexspec* [, *progress_proc*]]

Purpose

To reindex a file (rebuild an index).

Arguments

filespec is the file to process.

indexspec is the index to rebuild. If not given, all indexes are rebuilt.

progress_proc is a procedure to call to report progress on the operation.

Use

reindex is used to rebuild an index after it has become out of date. An index would become out of date if records were written to it using **save** rather than **update** or **create**. If large numbers of records are being added, it is generally faster to use **save** to write each one, then reindex the file afterwards.

The progress function is called regularly and is passed a single integer parameter. This is a number between 0 and 100 and gives the 'percentage complete' for the operation. The progress function may use the **delay** command to display an hourglass, or might display this percentage in a card on the screen using **card update**.

Progress functions are described in detail in the Tutorial. Note if you use an integer rather than a label for *indexspec* that indexes are numbered from 0 like all other resources.

Example

Here is a procedure to reindex a file, displaying an hourglass while doing so.

```
def proc progress (pc)
   delay pc
enddef

def proc reindex (curfile)
local i
```

```
delay on
ffor i=0 to @indexcount pupil - 1
   reindex pupil, i, progress
next
delay off
enddef
```

repeat

Syntax

repeat

...

until ***expr***

Purpose

To execute a set of statements until a terminating condition becomes true.

Arguments

expr is the terminating condition.

Use

This command starts a **repeat...until** loop. Statements following the **repeat** are executed up to the **until** statement. The expression ***expr*** is then evaluated, and if false (0), the statements within the loop are executed again. This continues until ***expr*** evaluates to a non-zero value.

Example

```
| find 10 * 10 the slow way
i=0
repeat
   i += 1
until i = 10*10
print "10 * 10 = ",i
```

See also

until, while, for, ffor

report dispose

Syntax

report dispose ***handle***

Purpose

To dispose of a report after use.

Arguments

handle is the handle of the report to dispose.

Use

This command should be used to dispose of a report after it is finished with. It first does a **report hide**, then removes the report from memory.

Reports must be disposed of after use; it is not advisable simply to hide them from view.

Example

See **report new** for an example.

See also

report hide, report new

report exit poll

Syntax

report exit poll ***handle***

Purpose

To stop polling a report.

Arguments

handle is the handle of the report to stop polling.

Use

This command will cause a **report run** command to terminate when it next checks the report. Control will return to an S-Base application which has issued a **report run** command on report ***handle***.

Unlike **report exit run** this command causes the report to immediately exit the **report run** without giving the user a chance to view the final page of the report. It is useful when something has gone wrong and report must be stopped altogether.

See also

report exit run

report exit run

Syntax

report exit run ***handle***

Purpose

To stop generating a report.

Arguments

handle is the handle of the report to stop.

Use

This command will cause generation of a report to stop. S-Base will assume that it has reached the end of the report. If the current page

contains any entries, the user will be able to browse as usual, but cannot generate any more pages. S-Base will not call the *request entry* handler again.

When the user exits the report, S-Base does a **report exit poll** and control will return to after the **report run** command which started the report generation.

See also

report exit poll

report hide

Syntax

report hide ***handle***

Purpose

To remove a report from the screen.

Arguments

handle is the handle of the report to remove.

Use

This command can be used after creating and displaying a report. It closes and deletes the window containing the report, thus removing it from the screen. **report show** will bring it back again.

This command does not dispose of the report — see **report dispose**.

Note that the report will not disappear until the Window Manager is next polled.

Example

See **report new** for an example.

See also

report new, report dispose, report show

report new

Syntax

report new ***handle,format*** [**,** ***title***]

Purpose

To create a new report.

Arguments

handle is an integer variable into which the handle for the new report will be placed.

format is the format to use for the report. This can be a string (format name) or an integer (a format number).

title is the title of the report, displayed in the title bar of the window containing the report.

Use

This command is used to create a new ready-to-run report for the display of information. The report has a format which specifies the elements of the report, the page size, etc.

A report consists of a series of pages, each with an optional header and footer and any number of columns. The columns have a number of *entries* running down them, and may also have a header and footer. S-Base draws entries in a column until it is full and then moves on to the next column. When all columns on the page are full, S-Base continues in the same way on a new page.

Format areas

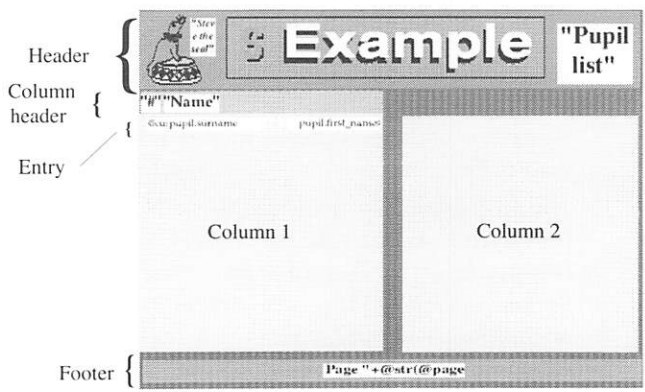
All format elements have an *area* attribute which allows S-Base to decide which area an element is in. The area attribute is important, since it determines how S-Base will draw the element on the page. For example, elements in the *entry* area are drawn in every report entry. Elements in the *page* area are drawn only once per page.

The various areas are listed below in the order that S-Base evaluates and draws them. Some area types can be included more than once: there can be several columns, and several different types of entry, for example.

Type	Purpose
Page	Defines the size of the image which will be printed. This area is generally the same size as the paper (less margins) for which the report is intended. If it is smaller, S-Base will centre the image on the paper. Elements in this area appear once on each page and will be drawn first. All elements are positioned relative to this area.
Page header	Defines the header which will appear at the top of each page. Elements in this area will appear once on each page.
Column	Defines columns for the entries to be drawn in. There can be several columns in a format.
Column header	Defines the header for each column. This can appear only once on the page. S-Base will automatically repeat the header for each column

	it draws.
Entry	Defines a report *entry*. There can be any number of different entry areas in a report. Elements within an entry area must follow the area and precede the next. Most reports have only one entry area.
Column footer	Defines the footer for each column. This can appear only once on the page. S-Base will automatically repeat the footer for each column it draws.
Page footer	Defines the footer which will appear at the bottom of each page. Elements in this area will appear once on each page.
Special	This area has no use in reports. Elements placed in this area will not appear on the report.

In general a report format has a layout similar to the diagram below. You can see header and footer areas which will appear at the top and bottom of each page. You can also see the two columns where report entries go. Column 1 has a column header which S-Base will automatically place above column 2. The report entry is at the top of column 1. This will be repeated down column 1 and then down column 2. S-Base will then create a new page and place entries down each column of that. The process continues until there are no more entries to place as signalled by **report exit run**.



To create a new entry in a report S-Base simply decides where to put it and evaluates all the elements in the entry area. But it first calls a *request entry* handler to tell the application which entry it is about to generate. This handler should load any information required for the entry and return. For example, the *request entry* handler for a customer list might load in the next customer. S-Base provides a function, **@entry**, which returns the number of the entry about to be generated. Entries are numbered from 0.

S-Base always requests entries in sequential order. That is, entry 5 can never be requested before entry 4. However, S-Base is free to request an entry more than once. It may do this at the end of a page. You should not assume that every new request is for a new entry. A function **@lastentry** is provided to find out which entry S-Base requested *last* time. You can compare this with **@entry** to decide whether to load the next entry.

Entry size

The size of an entry in a report is determined by two factors. Firstly, the width is set by the column containing the entry. An entry which is wider than the column containing it will protrude outside the column and may encroach on other columns. Secondly, the height is set by the entry area element used to generate the entry. You should ensure that the entry area element is large enough to contain all entries in the area. If not, entries in the report may overlap.

With text fields it is difficult to be sure that the field width you have decided on will be wide enough. Since S-Base normally truncates text at the right edge of a field, you may find that your report has parts missing where, for example, the video title is longer than the average length.

Obviously you can fix this by making all fields very wide — wide enough to hold a line full of @ characters for example. However, this wastes valuable column space. Another solution is to make the field deep enough to hold several lines of text. This reduces the column width but means that most entries are longer than they need be, with a large amount of white space in the report.

A better solution is to set a default height for the entry but allow it to expand where necessary. Set the height of the entry area element to the minimum height required. To do this, set the 'floating baseline' flag in all elements in the entry. This means that S-Base will drop the baseline of each element as required. If the element fits in one line, S-Base will not extend the entry at all. If the element requires three lines of text, S-Base will allow three lines of text, and extend the entry by two lines.

You can give a floating baseline to any element whose size can be calculated. This includes lines, rectangles, sprites and drawings. S-Base will ensure that the whole of the element is visible and that the entry accommodates it.

The order of format elements

The format layout editor, like all other places where formats are displayed, displays format elements in order from first to last. This means that for overlapping elements, the first one will be underneath the last one.

Because of this it may be necessary to ensure that area elements come before the elements which need to appear in that area. Otherwise the area element may obscure those elements. You can re-order format elements using the tools provided in the format layout editor, or by re-ordering elements directly in the element list window.

However, the only situation where format elements *must* appear after their area is where you are defining an entry. In this case you must follow the entry area with the entry elements. If the entry elements come before the entry area, S-Base will generate empty entries.

Text only reports

Some reports consist of a large number of pages. Others need to be printed out on multi-part stationery with a dot matrix printer. To help with these requirements, S-Base provides *text only* reports. There is a 'text only' button available in the 'format info' dialogue.

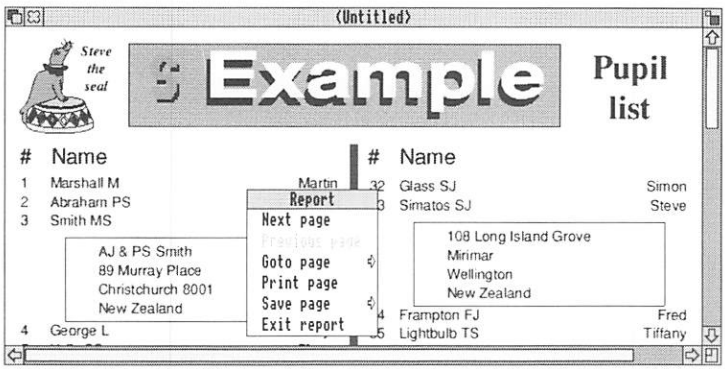
These can only contain text in the standard 'System font' (6.4 by 12.8 points with 12.8 point leading). Drawings, sprites, lines and rectangles cannot be placed in a text only report. These reports can be printed to a dot matrix printer at full speed using the printer's native fonts.

The report will still be printed through the RISC OS printer manager, but as text not graphics. No printer highlights are available in this mode. You cannot print bold or underlined text even though your printer may support it. Text only reports are for bulk printing without any embellishments.

Obviously the restrictions outlined above mean that it is not possible to achieve the same quality and complexity as with normal reports; but where speed is more important tham quality, text only reports are invaluable. S-Base reduces the format to a text grid, mapping each character of the format into one grid space. To obtain correct results you must ensure that all elements of the format are aligned with the text-only grid (the grid dialogue has a 'text only' button).

User control

After S-Base has generated a page it will allow the user to browse it before generating the next page. In this mode the user will be able to scroll around the page and look back at previous pages.



The user may also decide to print this page or save the page as a draw or text file. Text only reports are saved as text files; normal reports as draw files. Both types of file can be edited in their respective applications: !Edit for text files and !Draw for draw files. This can be useful for adding extra graphics to report pages before printing, although since S-Base formats can include draw files there is no reason why S-Base should not produce the finished result. Drawings can be imported into many applications, including Desktop Publishing software.

Multi-entry reports

It is usual for a report to contain only one type of entry. However, in the report above you can see that under some entries there is an address. This address is an entry just like the normal entry, but it appears only when the application desires it.

In fact, reports can have any number of entry types. Each has an *entry area* element, which contains the elements which go in the entry. The elements making up the entry must immediately follow the entry area element, and precede the next entry area element.

S-Base uses the first entry area it finds as its *default entry*. Under normal circumstances the whole report will consist of entries of this type. However, the application can direct S-Base to use a different entry area by use of the **report set entry** command. This allows the application to specify the entry area which will be used for subsequent entries in the

report.

S-Base will continue to use the new entry until it is told to change. In the report shown above, this processing is performed in the report's *request entry* handler. The application checks to see if the pupil about to be displayed has an address. If so, it sets the entry area to the 'address' entry. Next time S-Base generates an entry it will use the 'address' entry, and so produce the boxed address shown above. The 'address' entry has a different *request entry* handler, which simply sets the entry area back to the default. Thus after generating an entry containing the address, S-Base will go back to the default entry type.

This same approach can be used to generate complex reports with many different entry types. However, you should note that each entry created has a unique number, and entries are numbered regardless of which entry type is being used. For multi-entry reports, you cannot rely on **@entry** to give you the record to load in a *request entry* handler; you must use your own variable which you increment as required.

Automatic report generation

You will normally find automatic report generation adequate for creating simple reports. It can also be useful for creating the basis for a complex report. You should find that all you need to do is write a *request entry* handler (which is often only one line long) and put the **report new...report dispose** commands somewhere in your application.

You can use the format layout editor to add lines and rectangles to your reports to make them look more presentable. Logos can be added as sprites or drawings.

Automatic report generation can be used to produce text only reports. You should specify the system font style for all parts of the report. You may have to adjust elements and snap them to the text only grid.

See the Tutorial for more information on automatic format generation for reports.

Example

```
report new c, "pupil list", "Pupil List"
report show c
report run c
report dispose c
```

See also

report dispose, report run, report set entry

report run

Syntax

report run ***handle***

Purpose

To generate a report.

Arguments

handle is the handle of the report to generate.

Use

This command is used to start the generation of a report. The report must have been created with **report new** first.

S-Base generates all entries in the report one by one, placing them down the columns on each page. After each page is generated the user will have the opportunity to browse the page, print it out, etc. and then continue generating the report or exit.

S-Base will call the entry area's *request entry* handler for each entry in the report. The handler should set up the elements in the entry area. S-Base will then evaluate the elements to produce the entry. The handler may inspect **@entry** to decide which entry to load. **@lastentry** can also be used to find out whether S-Base is repeating an earlier request.

Example

See **report new** for an example.

See also

report new, **@entry**, **@lastentry**

report set entry

Syntax

report set entry ***handle*** [, ***entry***]

Purpose

To set the entry area that S-Base will use to generate the next entry.

Arguments

handle is the handle of the report.

entry is an *entry* area element.

Use

Most reports will need only one type of entry which will be repeated from top to bottom. More complex reports may require two or more different entry types.

For example, the video catalogue report allows the video title to be

followed by additional information about the title. Here there are two entry types: the basic title information, and the additional information.

S-Base must be told which entry type to use next. By default it uses the one it used previously. By default, when the **report run** command starts, this is the first *entry* area in the format.

By using this command in the *request entry* handler, you can direct S-Base to use a different entry area for the next entry.

Without an ***entry*** parameter, **report set entry** sets the next entry type to the default entry (the first *entry* area in the format).

Example

This example illustrates a *request entry* handler where there are two entry types: the pupil name and the address. The address is only printed if it exists and the user has requested it.

```
{ report area entry handler - loads the correct pupil
record }

{ check for a repeated record - if not, load the next
one }
if @entry <> @lastentry then rec = next_rec

| if there are no more records, stop generating
if !rec
   report exit run r
else
   | get the next record
   if rec = next_rec then next_rec = @next (curfile,
curindex, rec)

   | load this record
   load pupil, rec

   { change the entry to 'entry2' if we want to show the
address }
   if show_address && @len @trim pupil.duplicate_address
[0]
      report set entry @handle, "entry2"
   endif
endif
```

See also

report new

report show

Syntax

report show ***handle***

Purpose

To display a report in a window on the screen.

Arguments

handle specifies the handle of the report to display.

Use

This command should be used after creating a report. It creates and opens a window containing the report and displays it on the screen.

Note that the report will not appear until the Window Manager is next polled (with **report run**).

Example

See **report new** for an example.

See also

report new, **report run**

run

Syntax

run ***program_name***

Purpose

To execute a program.

Arguments

program_name is a string giving the name of the program to run.

Use

This command is used to run programs. The program name must give the full path of the program from the top level programs resource, with a dot (.) between each component. For example, a program called `video` in the top level resource could be run using `run "video"`, whereas a program within a program list called '`main`' would be run using `run "main.video"`.

Example

This example runs a program called `video_catalogue` in a root-level program list called `reports`.

```
| run the video catalogue report
```

```
run "reports.video_catalogue"
```

save

Syntax

save ***filespec, recnum*** [, ***real_filespec*** , ***bytecount***]

Purpose

To save a record from a file's record buffer to disc, without updating indexes.

Arguments

filespec is the file to which the record is to be saved.

recnum is the record number to save to.

real_filespec is the file from whose record buffer the data is to be saved. If not specified, it is taken to be the same as ***filespec***.

bytecount is the number of bytes to save. If not specified (so that ***real filespec*** is not given either), it is taken to be **@recsize** ***filespec***.

Use

This command is used to save records on disc. Saving to a file with fixed-length records generally involves specifying only the first two parameters. Unlike **update**, no indexes are updated, thus this command cannot be used to update indexed fields. This can cause the indexes to become out of date, requiring a reindex. Sometimes **save** is used for bulk updating and the file is then reindexed.

For variable length records, the application has the option of specifying both the source of the data (which may be from a different record buffer than the default ***filespec***) and the number of bytes of data (which may depend on the record type).

This is useful when saving different transaction types to the same transaction file and when saving different sized transactions to the same file. Sometimes the application needs full control over the save process.

In general, however, S-Base can get things right by using the 'zero length structure' and 'variable length field' attributes of a file. The example below illustrates this.

This command will update a file's record count if ***recnum*** is greater than **@recordcount** ***filespec***

Example

```
| Alter the plant name
load plant,50
plant.name = "Azelia Star Trek"
```

```
| save the change without updating indexes
save plant,50
```

The example below creates a transaction and saves it. Note that the **create** command would more usually be used for this. S-Base uses the ***num_items*** field to work out the size of the record. Here S-Base saves a record containing 3 items:

```
tran.type = TRANT_INVOICE
tran.num_items = 3
... (set up items)
save tran,5
```

See also

create, delete, load, update

select

Syntax

select [**update**] ***filespec***, ***indexspec***, ***proc*** [, {***value***}] [**for** ***range***]

Purpose

To find all records matching given search criteria.

Arguments

filespec is the file to search.

indexspec is the index to use for the search.

proc is a procedure to call with each record number found. This can also be given as a string containing the procedure name.

value is a value to use for the search.

range is a string containing an additional expression which must be true (non-zero) for all found records.

Use

select is used to search for records matching given criteria. It will search an index for all records matching the search fields given.

The search fields should be given in the same order as fields in the index, and the maximum number of search fields allowed is equal to the number of fields in the index.

String fields will be matched up to the length of the specified search string. Thus specifying an empty string will always match a string field.

If no search fields are specified, **select** will return all records in the file. Records are always returned in the order given by the index, so this is one way of processing all the records in a file in a particular order.

If **select** cannot find any record matching the criteria, or if there are no records in the file, it will never call the procedure.

The procedure ***proc*** will be passed one parameter: the record number of a matching record. It should process this record as it wishes then return an integer value. This value tells S-Base whether to continue with the search or to stop: 1 to stop, 0 to continue.

To process all records in a file ***proc*** should always return 0. Sometimes you will want to stop the search early, perhaps because you want to impose a limit on the number of records found. In this case, return 1 when you want the search to stop.

Filters

The ***range*** string provides an additional constraint on the search. This is a string containing an expression which S-Base will evaluate for all records which match the normal criteria. If this expression evaluates to true (non-zero), S-Base will pass the record to the handler procedure. If not, the record will be rejected.

There are some restrictions on the expression which can be placed in ***range***. Normally the expression will contain a simple comparison with a value, e.g. `customer.balance > 0`. S-Base will scan the expression to determine whether all the fields mentioned in ***range*** are contained within the index. For example, if the index contains only a video title field, then S-Base will discover that `customer.balance` is not included in the index.

If S-Base decides that all fields mentioned in ***range*** are included in the index, it can perform the search rapidly, without loading any records from the data file. If some other fields are mentioned, then S-Base will have to load each record from the data file to decide whether it matches the criteria. Note that S-Base will still do this fairly rapidly, but not as quickly as if the index contained all the information.

The restriction mentioned above relates to network operation. When an S-Base station needs to perform a search it simply passes it down the network to the S-Base server. The server then evaluates ***range*** for each record in the file and returns a list of record number to the station. The expression in ***range*** must be able to be evaluated by the S-Base server, which has none of the execution context available in the station. For example, no variables are defined in the server, and the values of fields in other files cannot be relied upon.

The example filter `customer.balance > 0` is fine, since no variables or other files are mentioned. However, something like `customer.balance`

`> total + transaction.limit` is doubly wrong. Firstly, it relies on the value of a variable called `total` which will not exist in the server; secondly it relies on the value of `transaction.limit`, which cannot be determined by your application.

Because of this restriction you should only put actual values in ***range***. In the example above, `total + transaction.limit` should be evaluated and placed in ***range*** using **@str**.

Record number ordering

It is possible to specify ***indexspec*** as −1. This means to use the normal record number ordering of a file. No values may be specified in the search (since there are no values in the 'index') but a range may be given.

Updating

If **update** is given in the **select** command, S-Base will update ***filespec***'s record with each record before calling ***proc***. The record will contain the values of all the fields contained within the index. This feature does not operate over a network.

For example, take an index on a `name` and `age` field in an address file. Imagine **select** is used to search for a name of 'Smith', with no age given, and it finds 10 matches (e.g. 'Smith AJ', 'Smith BM', etc). **select** will pass the record number of the first 'Smith' in the file to the handler procedure ***proc***. If **update** is given, S-Base will also update the address record with the name and age of the record found. The full name stored in the record would be stored in `address.name` and the age in `address.age`.

Updating is often useful to speed up the processing of a file. If all the information you need to process is stored in an index, you need not load each record from the data file. Just specify **update** in the **select** function and the data will be loaded quickly from the index. You can then use it in the handler procedure without loading from the data file.

Note that if you specify a range such that S-Base must load each record from data file, then your handler procedure will have all information available to it, since the entire record will have been loaded.

Example

Within the Video system the `title` file has an index on the `title` field. These examples illustrate the use of **select** for searching this index.

```
def proc dorec (rec)
print rec:-4, title.title
return 0
```

```
enddef
```

```
select update title, "title", "dorec"
```

Produces a list of all titles in order of title:

```
8   Aliens
6   Blackadder Goes Forth
5   Cinderella
4   Ghost
7   Life of Brian
2   Raiders of the Lost Ark
1   Star Wars
3   Terminator 2
10  The Reason Why
9   Word War II
```

Notice that the record need not be loaded with a **load** command in the procedure, since **select update** will automatically update the file's record buffer with the title from the index.

Changing the **select** command to:

```
select update title, "title", "dorec", "T"
```

will produce only records whose title starts with a 't':

```
3   Terminator 2
10  The Reason Why
```

Changing the **select** command to:

```
select update title, "title", "dorec" for "@trim
title.rating = ""G"""
```

will produce only records with a G certificate:

```
5   Cinderella
1   Star Wars
9   World War II
```

The same search can be done with index −1, giving the records in order of record number rather than title:

```
select update title, -1, "dorec" for "@trim title.rating
= ""G"""
```

```
1   Star Wars
5   Cinderella
9   World War II
```

See also

@find, @findn

selector clear protect

Syntax

selector clear protect ***handle***, {***entry***}+

Purpose

To unfade selector entries.

Arguments

handle is the handle of the selector to alter.

entry is an entry to unfade (numbered from 0).

Use

This command unfades selector entries after they have been created. Unfaded entries can be selected by the user.

See also

card new, selector set protect

selector delete

Syntax

selector delete ***handle***, ***start*** [, ***count***]

Purpose

To delete entries from a selector.

Arguments

handle is the handle of the selector.

start is the first entry to delete (numbered from 0).

count is the number of entries to delete (1 by default).

Use

This command is useful where a selector is used to display things that the user can delete. It can also be used to destroy all entries in a selector and start again.

The selector will shrink automatically to compensate for the lost entries. However, this command does not cause the selector to be redrawn. You should use **card update** (giving the selector's element label) to do this.

Example

```
{ handler for a click on a card with a selector with a
list of entries which the user can modify }
local s=@elementhandle (c, "list")
```

```
case @clickelementlabel

| handle a click on the 'delete' button
when "delete"
   | delete the currently highlighted entries
   local i, sel=@selectedhandle s

   { use the 'selected' bitstring to find out which
entries are selected }
   ffor i = @entrycount s - 1 to 0 step -1
      if @bittest (sel, i)
         | if this entry is selected, delete it
         selector delete s, i   .
         bit clear sel, i
      endif
   next

   | update the selector to reflect the changes
   card update c, "list"

   { fade the 'delete' button since no entries are now
selected }
   card set protect c, "delete"
...

endcase
```

See also

@selectedhandle, card update

selector insert

Syntax

selector insert ***handle***, [***before***,] {***string***}+

Purpose

To insert entries into a selector.

Arguments

handle is the handle of the selector.

before is the existing entry to insert these new entries before (numbered from 0). By default they are inserted at the end.

string is a string to place in an inserted selector entry.

Use

This command allows you to add new entries to a selector. An entry is just a string displayed in the selector. Each entry must be given in a separate ***string*** parameter. The selector will grow automatically to accommodate the new entries. However, this command does not cause the selector to be redrawn. You should use **card update** (giving the selector's element label) to do this.

This command must be used on a created selector for which you have a handle. Since selectors cannot exist on their own, this means you must have obtained the handle from a card using **@elementhandle**.

A value of -1 for ***before*** has the same effect as leaving out the ***before*** parameter: the new entries are inserted after all existing entries.

Example

```
| add some items to a selector called 'list' in card 'c'
local s = @elementhandle (c, "list")

selector insert s,"Entry 0", "Entry 2"
selector insert s,"Entry 3"

| insert before entry 2
selector insert s, 1, "Entry 1"
```

See also

@elementhandle, **card update**

selector set

Syntax

selector set ***handle***, ***entry***, ***string***

Purpose

To alter an entry in a selector.

Arguments

handle is the handle of a selector.

entry is the entry number to alter (numbered from 0).

string is the new string for the entry.

Use

selector set is useful for altering the text of an existing entry in a selector rather than deleting and recreating the entry. This command cannot be used to create new entries.

selector set does not cause the selector to be redrawn. You should use

card update (giving the selector's element label) to do this.

There are no restrictions on changing the length of an entry string.

Example

The following example is another extract from the handler mentioned in **selector delete**. It handles an 'edit' button, which allows the user to edit a single selected entry.

```
{ handler for a click on a card with a selector with a
list of entries which the user can modify }
local s=@elementhandle (c, "list")

case @clickelementlabel

{ handle a click on the 'edit' button. Note that this
button is protected unless the user has selected exactly
one entry from the selector }
when "edit"
   { edit the currently highlighted entry (there can be
only one }
   local sel=@selectedhandle s, ok=0, edit

   { use the 'selected' bitstring to find out which
entry is selected }
   e = @bitfirstset sel

   | open up a card allowing the user to edit this
   edit_entry = @entrytext (sel, e)
   card new edit, "edit selector entry"
   card show edit
   card poll edit
   card dispose edit

   { the click handler for edit's 'OK' button will have
set 'ok' to 1 if the user clicked ok. In this case we
must update the selector with the changes }
   if ok
      | change the selector's entry to the new value
      selector set s, e, edit_entry

      | update the selector to reflect the changes
      card update c, "list"
   endif
```

```
...

endcase
```

See also

@entrytext, selectedhandle, card update

selector set protect

Syntax

selector set protect ***handle***, {***entry***}+

Purpose

To fade selector entries.

Arguments

handle is the handle of the selector to alter.

entry is an entry to fade (numbered from 0).

Use

This command fades selector entries after they have been created. Faded entries cannot be selected by the user.

See also

card new, selector clear protect

set ptr

Syntax

set ptr ***handle, ptr***

Purpose

To set the file pointer of a file.

Arguments

handle is the file handle.

ptr is the position to point to in that file (measured in bytes).

Use

This command allows you to access files in a non-sequential manner. You can move the file pointer anywhere you wish but you may not move it outside the file.

Example

```
s:fd = @openout "testdata"
s:archive fd, 12
s:print @ptr fd
4
s:set ptr fd, 0
s:import fd, x
```

```
s:print x
12
```

See also

@ptr, @ext

set ext

Syntax

set ext ***filefield, size***

Purpose

To set the number of bytes stored in an external file field.

Arguments

filefield is the external file field to alter.

size is the new size for the file field.

Use

set ext can be used to set the amount of data held in an external file field. Any previous contents of the field are lost, and a new block is allocated on the disc heap.

This command is useful when an application wants to load data into the disc heap itself, without using the usual **import** or **buffer save** commands.

Example

```
{ make the photo field ready to receive 10000 bytes of
data }
set ext pupil.photo, 10000
```

See also

buffer save, import, @ext, @ptr

sort

Syntax

sort [**by** ***direction***,] ***main_array*** [: ***count***] [, {***var*** [* ***elements***]}+]

Purpose

To sort array variables.

Arguments

direction is an integer giving the sort order: 1 for ascending, -1 for descending. If not present, ascending order (direction 1) is assumed.

main_array is the array containing the items to be sorted.

count is the number of elements of ***main_array*** to sort (all of them by default).

var is an array to be sorted into the same order as ***main_array***.

elements is the number of elements of ***var*** which correspond to one element of ***main_array*** (1 by default).

Use

sort will sort ***count*** items (of any type) in the array ***main_array***. Any number of secondary variables can be sorted using ***main_array*** as the key, which is indicated by listing them at the end of the command.

A further option is provided for when a single element of ***main_array*** corresponds to many elements of a secondary array. For example, ***main_array*** may be a list of video titles and ***var*** a two dimensional array with a list of four prices (representing different discount levels) for each video title. Here you would set ***elements*** to 4.

Note that the sort order is determined only by ***main_array***. Elements of other arrays are simply swapped around in the same way as those of ***main_array***. There is no way to make any of the ***var*** parameters influence the sort order.

Example

In this example we have a list of video titles `title_list` and an array of prices called `prices` which need to be sorted:

```
| setup the variables
local title_list, prices, count

dim s title_list [@recordcount title]
dim i prices [@recordcount title] [4]

| get a list of titles
count = proc get_title_list (title_list, prices)

| sort the titles and prices arrays
sort title_list:count, prices * 4
```

The `4` in the last line matches the extra subscript of the `prices` array. It indicates that for every element of the `title_list` array, there are four elements of the `prices` array. To sort the `title_list` array (but ignore the prices array), you could use

```
sort title_list:count
```

The count is given since `title_list` may contain more items than `count` and the value of these items is likely to be the empty string `""`. Sorting the whole array would result in a large number of empty strings at the start of the array, followed by the ordered `count` items. We would then have to move all the ordered items to the start of the array, or process them where

they are. Either way, **sort** would take much longer than necessary.

sys

Syntax

sys *code* [, [*r0*], [*r1*] ... , [*r8*]] [**to** [*r0_var*], [*r1_var*] ... , [*r8_var*] [; *flags_var*]]

Purpose

To execute RISC OS system calls.

Arguments

code is the SWI code you wish to call. This can be given as either a string (eg "OS_WriteC" or a number (eg &400c0).

r0, etc. are integer values for the ARM registers. You can give values for r0–r8.

r0_var, etc. are integer variables into which the values of the ARM registers will be placed when the system call returns. You can give variables for r0–r8.

flags_var is an integer variable into which the flags word will be placed when the system call returns.

Use

This command gives access to the SWI calls available in RISC OS. These calls are in the RISC OS Programmers' Reference Manual.

As shown by the syntax above, you can omit arguments but the commas must be present to act as place holders for missing arguments. S-Base will pass 0 instead. You can omit 'trailing' commas.

You can use **@addr** to obtain the address of variables for passing to this call. Note that if you pass the address of a string variable, the SWI must not extend this string. In fact, any alteration of S-Base workspace in this manner will cause serious problems. Such problems are generally very difficult to track down.

Since S-Base provides a large selection of display and input facilities, you should find little use for the **sys** command.

It is good programming practice to isolate **sys** calls (and other low-level or external operations) into procedures, perhaps in a single procedure resource, so that they do not appear throughout the main application code.

Errors returned from the SWI are dealt with in the same way as other errors. However, if you set the 'X' bit (&20000, or use, e.g. `sys "XOS_SpriteOp"` rather than `sys "OS_SpriteOp"`) then errors will not cause your application to stop execution. You can use **@errnum** to check

for errors returned in this case.

Example

This example uses **sys** to increase the memory available to the sprite area by 16k.

```
sys "OS_ChangeDynamicArea", 3, 16384 to ,moved
if moved < 16384 { may be 32768 on 4MB machines }
   error 1,"Cannot allocate sprite space"
endif
```

See also

buffer new, @addr, @stringize

table clear protect

Syntax

table clear protect ***handle, entry,*** {***element***}+

Purpose

To unprotect elements in a table entry.

Arguments

handle is the handle of the table to alter.

entry is the entry containing the elements to alter.

element is an element to unprotect.

Use

This command allows you to unprotect elements in a table.

Unprotected elements can be clicked on and can contain the cursor. See **table new** and **card new** for more details. Note that table elements will only stay unprotected (if protected in the format element flags dialogue) while they are in the cache; you should use this command in request entry handlers.

Elements protected using the format element flags dialogue can still be unprotected with this command.

Example

Protect the `name` field of the current entry.

```
table clear protect t, @entry, "name"
```

See also

card clear protect, table new, table set protect

table delete

Syntax

table delete ***handle, start, count***

Purpose

To delete entries from a table.

Arguments

handle is the handler of the table to alter.

start is the first entry to delete.

count is the number of entries to delete.

Use

table delete deletes a section of a table, updating the screen as required.

The table is not contracted after this operation — use **table ext** for this. You must also update any data structure used by the request entry handler so that when S-Base next calls the request entry handler (never during the execution of this command), the correct information will be returned.

Example

```
def proc delete_entry (t, e, return recnum)
{ delete an entry 'e' from a table 't' of records, the
record number of each entry being stored in recnum[] }

| first delete from table & contract the table
table delete t, e, 1
table ext t, -1

| now from the array
array delete recnum [e], 1
enddef
```

See also

table ext, table insert, table new, @ext

table dispose

Syntax

table dispose [update] ***handle***

Purpose

To dispose of a table after use.

Arguments

handle is the handle of the table to dispose.

Use

This command should be used to dispose of a table after it is finished with. It first does a **table hide**, then removes the table from memory.

If **update** is given in the command, the template used to create the window is updated with the table's position when last hidden. This means that the table will always pop up where the user last left it rather than always returning to its original position when recreated.

Tables must be disposed of after use; it is not advisable simply to hide them from view.

If a table is attached to a card, it must be detached (or the card destroyed) before the table can be destroyed.

Example

See **table new** for an example.

See also

table hide, table new

table exit poll

Syntax

table exit poll ***handle***

Purpose

To stop polling a table.

Arguments

handle is the handle of the table to stop polling.

Use

This command allows control to return from an executing **table poll** command.

S-Base marks the table ***handle*** as completed. If S-Base is currently waiting inside a **table poll** command for that table, then the **table poll** command will eventually return and execution will continue from that point.

Note that this command has no immediately discernible effect (it waits until the **table poll** command next looks at the table) and no effect at all if S-Base is currently polling another table or card, etc., even if that table or card's **table poll** is nested inside the **table poll** for table ***handle***. You should only use **table exit poll** on a table when you know that S-Base is currently executing a **table poll** for that table. **table poll** will execute until told to stop by a **table exit poll** for the same table.

Example

```
{ keypress handler for a table - exit when escape
pressed }

| check which key was pressed
case @key

| if <escape>, then quit the table
when ESCAPE
   table exit poll @handle

endcase
```

See also

table new, **table poll**

table ext

Syntax

table ext ***handle***, ***amount***

Purpose

To extend or contract a table.

Arguments

handle is the handle of the table to alter.

amount is the change in entry count. This is positive to increase the number of entries, negative to reduce the number.

Use

table ext is useful when an application must add entries to, or remove entries from, a table. S-Base will alter the number of entries in the table by the given amount. If the entry count is being reduced then entries will fall off the end.

S-Base will call the table extension handler as usual, so if this returns 0 (indicating the extension is not allowed) the extension will not occur.

See also

table new, **table set ext**, **@ext**

table hide

Syntax

table hide ***handle***

Purpose

To remove a table from the screen.

Arguments

handle is the handle of the table to remove.

Use

This command should be used after the user has finished with a table. It closes and deletes the window containing the table, removing it from the screen.

Note that the table will not disappear until the Window Manager is next polled.

Example

See **table new** for an example.

See also

table dispose, table new

table insert

Syntax

table insert ***handle, start, count***

Purpose

To insert entries into a table.

Arguments

handle is the table to alter.

start is the entry to insert the new entries before.

count is the number of entries to insert.

Use

table insert inserts entries into a section of a table, updating the screen as required.

The table is not expanded before this operation — use **table ext** for this. You must also update any data structure used by the request entry handler so that when S-Base next calls the request entry handler (never during the execution of this command), the correct information will be returned.

Example

```
def proc insert_entry (t, e, return recnum)
{ insert an entry before 'e' in a table 't' of records,
the record number of each entry being stored in
recnum[]. Does not expand 'recnum' itself }

| first expand the table and insert the entry
table ext t, 1
table insert t, e, 1
```

```
| now insert into the array
array insert recnum [e], 1

| zero the new array record number
recnum [e] = 0
enddef
```

See also

table ext, table insert, table new, @ext

table new

Syntax

table new *handle, format, entry_count*

Purpose

To create a new table.

Arguments

handle is an integer variable into which the handle of the new table will be placed.

format is the format (either a format label or number) to use for the table.

entry_count is the number of entries in the table.

Use

A table is a series of entries in a scrollable window, similar to a spreadsheet or an S-Base report. All entries are of the same basic structure, with the same fields, etc.. The user can edit fields just as with cards. In fact, since each entry looks like a card, a table can be thought of as a scrolling list of cards. You should be familiar with cards and reports before attempting to use tables, as they have much in common which is not all repeated here.

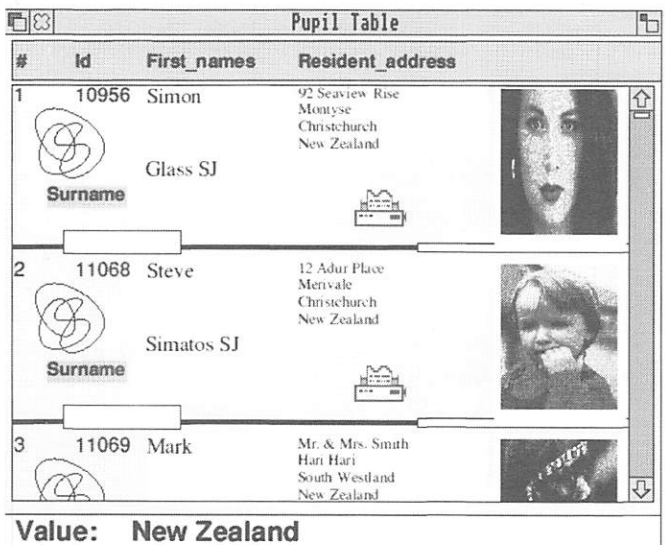
Most of the features of a table are controlled by its format. For example, you can set the size of the scrolling entry list (the pane), the header and footer information that goes above and below the pane, and the layout of entries themselves.

The table layout

Each table has a format which defines the look of the table. It specifies the elements which should appear in the table and which template to use for the table. The template editor is normally used to define the look of the window and any icons it has. You then attach the template to the format and set out the elements you want to appear in the window.

Tables can also have handlers attached to them. For example, you could instruct S-Base to call a handler every time the user clicks in a certain field. However, you should remember that only the elements in the table's entries are 'live'. You may put elements in the page header and page footer areas, but they will not respond to mouse clicks. The user will not be able to click on icons outside the entry area, nor will it be possible to position the cursor inside a field outside the entry area. You will often attach a card to a table so the user can control it.

A typical table is shown below. There is a main window which holds the header elements. Attached below the main window is the value bar, and in the middle of this is the entry pane. This pane contains all the entries in the table and it is this window that is the real 'table'. The cursor can be placed in one of the entries and it can be edited just like a card.



You will normally create a template for your table which defines the look of the window and any icons the window contains. In this example the main window has no scroll bars. Often you will include a row of icons allowing the user to control the table, although you will then have to attach the table to a card containing the icons.

Element types

Tables support almost all the types of elements which can be created in your format:

prompt	The expression is evaluated and displayed in the table. This text is fixed once the entry is created.
icon	Not supported.
field	The expression is evaluated and displayed in the table. The user may move the cursor around the fields and edit them.
selector	A selector window is created and displayed within the table. The selector is a separate entity from the table, but attached to it.
drawing	The drawing is displayed in the table.
sprite	The sprite is displayed in the table.
area	Not displayed. Used for grouping other elements.
line	Displayed in the table.
rectangle	Displayed in the table.

Format areas

All format elements have an *area* attribute which allows S-Base to decide which area an element is in and where to draw the element in the table. For example, elements in the *entry* area are drawn in every table entry in the entry pane. Elements in the *page header* area are drawn only once, outside the entry pane.

The various areas are listed below. Only one element of each type can be included. Unlike reports there can be only one column, and only one type of entry is allowed.

Type	Purpose
Page	Not used. Elements in this area will not appear.
Page header	Not used. Elements in this area will not appear.
Column	Defines the size and position of the entry pane. This should be wide enough to hold one entry, and preferably long enough to hold several entries.
Column header	Defines the header which will appear above the entry pane. You may put elements in this area.
Entry	Defines a table *entry*. There can be only one entry area in a table. The size of the entry area

	element defines the size of an entry. All entries are the same size.
Column footer	Defines the footer which will appear below the entry pane. You may put elements in this area.
Page footer	Not used. Elements in this area will not appear.
Special	Elements placed in this area will not appear on the report. This area is used for defining the value bar.

Table entries

A table consists of a number of entries, rather like a report. However, unlike a report, there is only one 'page', and the user can scroll from top to bottom simply by dragging the scroll bar. As with reports, S-Base draws an entry by simply deciding where to put it, and evaluating all the elements in the entry area. S-Base first calls a *request entry* handler to tell the application which entry it is about to generate.

This *request entry* handler should load any information required for the entry and then return. For example, the *request entry* handler for a table of customers might load in the required customer. S provides a function, **@entry**, which returns the number of the entry about to be generated. Entries are numbered from 0.

Unlike reports, S-Base can request entries in any order. In response to user action, S-Base may request entries 59 through 186 (one at a time) and then request entry 5. Your *request entry* handler must be able to supply the information requested whenever it is requested and two calls to the *request entry* handler with the same entry number should yield the same results.

To improve performance, S-Base caches entries in memory. For example, once S-Base has requested and cached entries 0 to 30, it will be able to handle all screen updates for such entries and the user will be able to scroll through those 30 entries without S-Base calling the *request entry* handler. As soon as entry 31 needs to be drawn on the screen, S-Base will request it.

The order of format elements

The format layout editor, like all other places where formats are displayed, displays format elements in order from first to last. This means that for overlapping elements, the first one will be underneath the last one. Note

that icons always appear on top of all other elements, as these are drawn by the Window Manager.

Entry size

The size of an entry in a table is determined by two factors. Firstly, the width is set by the column containing the entry (the column area element). An entry which is wider than the column containing it will protrude outside the entry pane and will not be visible. Secondly, the height is set by the entry area element used to generate the entry. You should ensure that the entry area element is large enough to contain all entries in the area. If not, entries in the table will overlap.

With reports you can give elements a floating baseline, so that they extend downward if required. However, in a table all entries must be the same height. Fields which do not fit in the available space will scroll as with cards. Other types of elements will be truncated, or run off the end of the entry area into the next entry.

Handlers

There is no built-in user control of tables. You must take care of this by adding click handlers to icons and keypress handlers to fields, etc, and possibly by attaching a card to the table which has controls on it. Once you have issued a **table poll** command for a table, that command will not return control to your application until a **table exit poll** is issued from one of the card's handlers.

S-Base supports the following handlers for tables open on the screen. You will normally use **@element** or **@elementlabel** to decide which element the event related to. **@entry** will give the entry number:

request entry	Called to request information about an entry in the table.
enter entry	Called when the cursor enters a new entry.
exit entry	Called when the cursor leaves an entry. You should save to disc any information in the entry which will be destroyed by loading another entry.
click	Called when the user clicks in the table. Use **@clickentry**, **@clickelement** and **@clickelementlabel** to find out what the user clicked on.
keypress	Called when the user presses a key in the table. Use

	@key to get the Window Manager key code (e.g. 13 is <return>). A list of key codes is given in Appendix A.
enter field	Called when the cursor enters a new field.
exit field	Called when the cursor leaves a field.
menu maker	Called when the user presses Menu in a table. Use **@clickelement** and **@clickelementlabel** to find out what the user clicked Menu over.
menu select	Called when the user selects an option from the table menu. Use **@clickelement** and **@clickelementlabel** to find out which element the user clicked Menu over, and **@menuselect** to find out which menu item was selected.

It is important to understand the order in which handlers are called. A **table poll** command will first cause the *request entry* handler to be called a few times so it can redraw the entry pane. S-Base will then decide where to place the cursor (say entry 0, field `name`). S-Base then calls the *request entry* handler asking for entry 0 (so that everything is set up), then the *enter entry* handler, also for entry 0. Finally S-Base calls the *enter field* handler for 'name'.

When the user moves the cursor to a new element in a new entry (say field 'name' of entry 1), S-Base will *first* decide where to place the cursor, *then* call the handlers. First it calls the *exit field* handler for 'name', then the *exit entry* handler for entry 0. Then it calls the enter entry handler for entry 0 and finally the *enter field* handler for 'name'.

It is important to understand that S-Base decides where to place the cursor *before* calling the handlers. Nothing you can do in any of the handlers mentioned above can change where S-Base is to place the cursor next. It has already been decided.

You may find the functions **@justaltered**, **@elaltered**, **@entryaltered** and **@altered** useful in some of these handlers. For example, use the *exit entry* handler to save the information the user has entered into an entry. By checking the value of **@entryaltered** before saving the data you can avoid unnecessary disc access.

@altered is useful where the whole table must be saved when the user has finished with it. If anything in the table has been altered, **@altered** will return 1. You need not bother to save the information if **@altered** returns 0.

Nested tables

The same notes regarding nested cards apply to nested tables. However, the **close** option of the **card poll** command does not have a counterpart in the **table poll** command. You must check for **@clickelementlabel** being `"<scared>"` in the table's click handler.

Often you will be controlling things from a scared card anyway, in which case the table can be closed when the card closes.

Automatic report generation

You will normally find automatic report generation adequate for creating simple tables. It can also be useful for creating the basis for a complex table.

You can use the format layout editor to add lines and rectangles to your tables to make them look more presentable. Logos can be added as sprites or drawings as with cards.

See the Tutorial for more information on automatic format generation for reports.

Evaluation and field width

The same notes regarding card evaluation and field width apply to tables. S-Base evaluates all expressions in all prompts, fields, icons, etc. in the entry area *when the table is created* (**table new**) and stores the values produced. If an element evaluated at this point was found to contain 12 characters, then the corresponding element in *every single* entry will be 12 characters long. Any attempt to enlarge the field after creation (even with **table update**) will fail.

For this reason, numeric expressions should normally be followed by a field width giving enough space for the maximum expected value. String expressions are similar, although file fields are padded with spaces to the maximum length anyway.

Selectors and Icons

Selectors and icons are not supported in table entries. You will seldom find the need for such complexity.

User control

Tables behave similarly to cards on screen. The user may move between entries and fields using the mouse and the cursor keys.

S-Base will endeavour to keep the cursor on the screen at all times, and

this may cause S-Base to scroll the entry pane to ensure that the whole of the current entry is visible (and preferably the entries on either side).

The following keys are understood and processed by S-Base before the keypress handler is called:

Key	Action
right arrow	Move cursor to the right one character within the current field.
left arrow	Move cursor to the left one character within the current field.
up arrow	Move cursor to the previous line within the current field. If already on the top (or only) line, move to the start of the nearest field above the current one. If already in the top field, move to the nearest field in the entry above. S-Base will not move the cursor if there is no previous entry.
down arrow	Move cursor to the next line within the current field. If already on the bottom (or only) line, move to the start of the nearest field below the current one. If already in the bottom field, move to the nearest field in the entry below. S-Base will try to extend the table is there is no next entry.
return	Move to the next field. If already in the last field, move to the first field of the next entry. S-Base will try to extend the table is there is no next entry.
tab	Move to the next field. If already in the last field, move to the first field of the next entry. S-Base will try to extend the table is there is no next entry.
sh-tab	Move to the previous field. If already in the first field, move to the last field of the previous entry. If already in the first entry, the cursor does not move
ctl-U	Erase the current field and move the cursor to the start of the field.

ctl-left arrow	Move the cursor to the start of the current field.
ctl-right-arrow	Move the cursor to after the last non-blank character in the current field.
delete	Delete the character to the left of the cursor and move the cursor back one character.
copy	Delete the character to the right of the cursor.

If the card has a close box and the user clicks on it, S-Base will call the format's click handler, as with cards, setting **@clickelement** to –2 and **@clickelementlabel** to `"<close>"`.

Within any handler, if there is no current field, **@element** will return –1 and **@elementlabel** will return an empty string. **@clickelement** and **@clickelementlabel** work the same way. If there is no current entry, **@entry** will return -1.

Automatic recalculation

This works as for cards. The *exit entry* handler will need to save the information entered by the user.

The value bar

The value bar is defined and operates in the same way as with cards. There is a corresponding **table update valuebar** command. You should ensure that the 'valuebar' field is placed in the 'Special' area.

Protected elements

These are supported as with cards, although the **table set protect** command differs slightly from the corresponding card command.

Moving the cursor

S-Base provides commands to allow the application to move the cursor within a field, or to a different field altogether (**table set cursor** and **table set element**). You can also move entries with **table set entry**. As with all such movement, S-Base notes the change but will not get around to moving the cursor until the machine is next idle.

Redrawing and the cursor

As has already been discussed, S-Base calls the *request entry* handler for an entry before placing the cursor in that entry. S-Base also calls the *request entry* handler when it needs to redraw an entry on the screen. But

S-Base must guarantee to the application that the record loaded corresponds to the entry containing the cursor, not necessarily the last entry drawn on the screen.

For example, in a video title table imagine the cursor is in the first entry (0) and the title is 'Return of the Jedi'. S-Base has called the *request entry* handler requesting this entry and it has been loaded. Now suppose the user scrolls the table downwards, so that S-Base has to redraw a different entry, say entry 15. S-Base must call the *request entry* handler requesting entry 15, which would mean that entry 0 is no longer loaded, even though the cursor has not moved.

The solution adopted by S-Base is to save the entry information before starting a redraw and restore it afterwards. In the above example, S-Base will save all text entered into entry 0 before requesting entry 15. Then when the redraw is complete S-Base will call the *request entry* handler for 0 again. This should restore the correct entry information minus any changes the user has typed in. Then S-Base 'types in' the changes saved before the redraw to restore everything as it was.

This process is usually transparent to the application. All the application need know is that S-Base will take care of saving and restoring the current entry. S-Base will call the *request entry* handler rather more often than the application might expect, but this by itself will not cause the application to fail. Since tables cache the most recently accessed entries, S-Base will not always call the *request entry* handler during the redraw save and restore process.

Attaching tables to cards

As mentioned above, tables cannot contain selectors or icons. However it is often useful to display a table where the user can enter data, and 'OK' and 'Cancel' to accept or reject the data. This cannot be done with a normal table.

The solution is to create a card containing the OK and Cancel buttons and to attach the table to it. To do this you create an area element in the *special* area and issue a **card set table** command instead of the usual **table show**. Now whenever you issue a **card show** command, the table will appear in the card over the area element. When you hide the card, the table will disappear too. This works just like attached selectors.

An attached table only has an entry pane. There is no need for a window behind this pane, since the card is holding the pane in place. Any elements in the page header and footer areas will not appear. You should copy these

elements onto the card format.

You cannot dispose of a table while it is attached to a card. You can detach the table with the **card clear table** command, or by destroying the card.

Table extension

Tables normally have a fixed number of entries. This is decided on before the table is created and the number of entries is passed to the **table new** command. In some situations it is desirable to allow the table to shrink or expand after creation. For example, a table where the user enters a transaction containing many items will expand as the user enters more items.

S-Base will sometimes decide that the table should be extended or contracted. For example, if the user tries to move beyond the last entry in a table, S-Base will try to increase the number of entries in the table. S-Base will also perform this operation when the **table ext** command is executed.

When S-Base wants to alter the number of entries in a table it calls an *extension* handler attached to the entry area (or the entire format) to find out whether extension (or contraction) is permitted. If no handler is provided, extension cannot occur. Within this handler, the function **@ext** returns the new extent (required number of entries). The handler should return true (1) if the extension should be allowed, else 0. The handler can also alter the new extent (perhaps to restrict the amount of change at any one time) using **table set ext**.

Tables which support extension often start with no entries. In this case, S-Base will call the extension handler to add a single entry during execution of the **table new** command (and you would not normally reject this request). At this point, the table has been allocated a handle but this has not yet been assigned to the variable in the **table new** command. For this reason you should always use **@handle** in table extension handlers to determine the handle of the table being extended, rather than assuming that the handle has been assigned to a variable. This is good practice for all handler types.

Inserting and deleting entries

S-Base provides facilities for inserting and deleting entries into/from a table. **table insert** will insert entries into a table and update the table on screen. **table delete** will do the same for entry deletion. You are responsible for ensuring that the *request entry* handler returns the updated information correctly after one of these commands has been executed. If

you store entry information in an array, you may find **array insert** and **array delete** useful.

Example

A standard table:

```
table new t, "pupil table", @recordcount pupil
table show c
table poll c
table dispose c
```

A table attached to a card. Note that the card is here assumed to have an area element called 'table' in the *special* area which marks where the table should be attached:

```
table new t, "transaction table", tran.num_items
card new c, "transaction editor"
card set table c, "table", t
card show c
card poll c
card dispose c
table dispose c
```

See also

table dispose, table poll, table show

table poll

Syntax

table poll [**until** ***eventcode*** ,] ***handle***

Purpose

To wait for the user to finish with a table.

Arguments

eventcode is the Window Manager event code to wait for before returning control to the program.

handle is the handle of the table to wait for.

Use

This command is useful after displaying a table on the screen with **table show**. It waits for a signal that the table is complete (e.g. the **table exit poll** command), then returns control to the program.

If ***eventcode*** is given, then S-Base will cease polling when it receives an event of that type (e.g. 0 for null events) for any window. This is most useful to allow primitive multitasking during monotasking operations – polling until a null event is received allows other applications some processor time. See Appendix B for a list of Window Manager event

codes.

Example

See **table new** for an example.

See also

table new

table set cursor

Syntax

table set cursor ***handle, position***

Purpose

To set the cursor position in the current field.

Arguments

handle is the handle of the table.

position is an integer giving the new position; 0 being the extreme left.

Use

This command is used mainly for setting the cursor to the start of a field after updating it. S-Base notes the change and will move the cursor when the machine is next idle.

The cursor position is measured in characters, with 0 being the start of a field. The position is always *between* characters. Thus 0 is before the first character and 20 is after the last character for a field of length 20. The cursor may not move outside the range [0...length of field]. S-Base will ignore any such request.

Example

```
| update the 'name' field and reset the cursor
table update c, @entry, "name"
table set cursor c,0
```

See also

table set element, table set entry

table set element

Syntax

table set element ***handle, element***

Purpose

To change the field containing the cursor.

Arguments

handle is the handle of the table.

element is the element which should gain the cursor.

Use

This command is useful when the cursor must be placed in a particular table, or in a particular field in the current entry. The cursor position within the element will be set to 0.

S-Base notes the change and will move the cursor when the machine is next idle.

Example

```
{ open a new table and put the cursor in the 'name'
element }
table new c, "title"
table show c
table set element c,"name"
table poll c
table dispose c
```

See also

table set cursor, table set entry

table set entry

Syntax

table set entry *handle, entry*

Purpose

To change the entry containing the cursor.

Arguments

handle is the handle of the table.

entry is the entry which should gain the cursor.

Use

This command is useful when the cursor must be placed in a particular entry in a table. The element will remain unchanged (**@element**) but the cursor position within the element will be set to 0.

S-Base notes the change and will move the cursor when the machine is next idle.

Example

```
| open a new table and put the cursor in entry 5
table new c, "title"
table show c
table set entry c,5
table poll c
table dispose c
```

See also

table set cursor, table set element

table set ext

Syntax

table set ext *handle*, *new_extent*

Purpose

To restrict the extension of a table from within a table extension handler.

Arguments

handle is the handle of the table.

new_extent is the new extent to which the table is allowed to expand.

Use

table set ext is only useful within an *extension* handler. When S-Base calls the handler to request an extension (or contraction) it sets **@ext *handle*** to the required number of entries.

If you want the extension to succeed, but S-Base is requesting too many entries, you can reduce the value of **@ext *handle*** using this command. S-Base will not allow you to specify a ***new_extent*** in excess of the original value.

Example

```
| Extension handler for a transaction table

local t = @handle

| don't allow more than 100 entries in the table
if @ext t > 100 then table set ext t, 100
return TRUE
```

See also

table ext, table new, @ext

table set handle

Syntax

table set handle *handle*

Purpose

To place the cursor in a table.

Arguments

handle is the handle of the table to receive the cursor.

Use

table set handle is useful when you want a particular table to claim the cursor, perhaps because the user has clicked on a particular icon. S will place the cursor in the table (without changing that table's current element or cursor position) when the machine is next idle.

S does a **table set handle** automatically in the **table poll** command.

Example

See **card set handle** for an example of a similar command.

See also

card set handle, table set cursor, table set element

table set protect

Syntax

table set protect ***handle***, ***entry***, {***element***}$^+$

Purpose

To protect elements in a table entry.

Arguments

handle is the handle of the table to alter.

entry is the entry containing the elements to alter.

element is an element to protect.

Use

This command allows you to protect elements in a table. Any element type in a table can be protected.

Protected elements cannot be clicked on and cannot contain the cursor. See **table new** and **card new** for more details. Note that table elements will only stay protected while they are in the cache: you should use this command in request entry handlers.

Example

See **card set protect** for a relevant example.

See also

card set protect, table new, table set protect

table show

Syntax

table show ***handle***

Purpose

To display a table in a window on the screen.

Arguments

handle is the handle of the table to display.

Use

This command should be used after creating a table. It creates and opens a window containing the table and displays it on the screen.

Note that the table will not appear until the Window Manager is next polled (e.g. with **table poll**).

Example

See **table new** for an example.

See also

table new, table poll

table update

Syntax

table update ***handle*** [, ***entry*** [, {***element***}$^+$]]

Purpose

To recalculate and re-display elements of a table.

Arguments

handle is the handle of the table to update.

entry specifies the entry to update. It must exist in the table.

element is an element of the table which needs updating. This may be given as either the element label (enclosed in quotes) or the element number.

Use

This command is useful when the application changes the values 'behind' the table. If you give an entry and a list of elements, S-Base updates only those elements on that entry. If you just give an entry then S-Base will update all elements in that entry.

If you do not specify an entry, S-Base will update the whole table. This is useful where the entire contents of the table have changed. For example, in a table which lists the transaction details for a customer you would want to update the entire table when the user selects a new customer.

Prompts cannot be updated with **table update**. A protected field should be used instead.

Note that S-Base will only redraw fields which have changed.

Example

```
! load a new customer into a table
def proc load_new_customer (t, num_records)
```

```
   | first change the number of entries in the table
   table ext t, num_records - @entrycount t

   { now update the entire table to show the new
transactions }
   table update @handle
enddef
```

See also

table new

table update valuebar

Syntax

table update valuebar ***handle*** , [***expr*** [***fieldspec***]]

Purpose

To display an updated value in the value bar.

Arguments

handle is the handle of the table to update.

expr specifies an expression to be converted to a string and displayed. If this is omitted, the normal expression is used.

fieldspec specifies how the string conversion will take place (see **print**).

Use

This command is useful when the application wants the value bar to show a value for elements which do not have auto-recalculation.

S-Base can never scan a *manual-recalc* field to produce a value. Therefore, when the cursor is in a *manual-recalc* field, the value bar will be blank. The application may insert its own value using this command.

If no expression is given, this command has no effect on *manual-recalc* fields. For *auto-recalc* fields, it just updates the value in the value bar (as would happen normally anyway).

Example

See **card update valuebar** for an example of how to update a value bar.

then

Syntax

if ***expr*** **then** ***command***

Purpose

To mark a single-line **if** statement.

Arguments

expr is an expression which determines whether ***command*** is executed.

command is any valid S command.

Use

This command forms part of the single-line **if** statement. See **if** for more details.

See also

if, else, endif

unlock

Syntax

unlock [***filespec*** [, ***recnum***]]

Purpose

To unlock files and records.

Arguments

filespec is the file to unlock. If not given, all files are unlocked.

recnum is the record to unlock. If not given, all records in the file are unlocked.

Use

unlock is used to lock a record, an entire file, or all files in the application. In stand-alone mode this command is ignored.

The station must own the relevant locks (i.e. must have called **@lock** first) in order to be able to remove them.

Example

See **@lock** for an example.

See also

@lock

until

Syntax

until ***expr***

Purpose

To specify the terminating condition for a **repeat...until** loop.

Arguments

expr is the terminating condition. The loop executes repeatedly until this becomes true.

Use

See **repeat** for details.

See also

repeat, while

update

Syntax

update ***filespec , recnum***

Purpose

To update an existing record in a file (and all indexes).

Arguments

filespec is the file containing the record to update.

recnum is the record to update.

Use

This command is used to update a record in a file. The record should previously have been created with the **create** command. Unlike **save**, this command updates all indexes.

Files with variable length records are handled in the same way as with **create**. It is also possible to extend a record (i.e. update it with a larger record) even if it is not the last record in the file. S-Base will copy the entire file as required (sometimes this may take some time).

Note that S-Base reads the record from disc to decide what has changed in the record being written. S-Base assumes that the index reflects the current record and should be updated to reflect the changes. If any index is not up to date, this command will fail.

Example

```
| update a plant record
load plant,50
plant.name = "New record"
update plant,50
```

use

Syntax

use ***filefield***

Purpose

To assign a value to a floating structure.

Arguments

filefield is the value to assign to the structure. The actual structure to

assign is given by the type of *filefield*.

Use

This command is only useful with floating structures. A floating structure is a file which does not exist on disc and has no memory allocated to it for a record. It floats around in memory and can be anchored anywhere.

Another file (a real disc file) may include this floating structure as one of its fields. The **use** command allows you to attach the floating structure to that field, or to an element of that field.

Assigning a value to a floating structure is called *anchoring*.

Note that a floating structure *must* be anchored before use, or you will get an error. You should never rely on the floating structure being anchored to the same place as last time. Always set it up just before using it.

Example

This example assumes a floating structure called `details` containing `name` and `price` fields. A file called `stock` has a field `list` which is an array of 100 `details`.

```
| process the details in the stock line
local i

for i=0 to 99
   | anchor 'details' to the correct array element
   use stock.list [i]

   | print the details
   print details.name, :2, details.price:10:2
next
```

The same code could be written without **use** like this:

```
| process the details in the stock line
local i

for i=0 to 99
   | print the details
   print stock.list [i].name:2, stock.list
[i].price:10:2
next
```

vdu

Syntax

vdu ***value*** { ; | , ***value*** } [;]

The ***values*** can be separated by a semicolon or a comma, not just a comma as the{ } suggests.

Purpose

To write a value directly to the VDU drivers.

Use

This command can be used to write a list of values directly to the VDU drivers (and generally to the screen). If ***value*** is followed with a semicolon (;) it is written as a 16-bit quantity (low byte first), otherwise it is written as an 8-bit quantity.

This command has limited use in a Wimp environment. You can use it to generate a short error beep (`vdu 7`). If you are controlling the printer directly you may wish to use `vdu 2` to turn the printer on and `vdu 3` to turn it off. Beware of using **vdu** commands in the command line, as redrawing of the command line window could be compromised.

Example

```
def proc handle_error (message)
   local c

   card new c, "error"
   card show c

   | beep as we open the error card
   vdu 7

   card poll c
   card dispose c
enddef

| plot a point (same as plot 69, 1000, 1000)
vdu 25, 69, 1000; 1000;
```

See also

plot

when

Syntax

when {*expr*}

Purpose

To indicate a new clause in a **case** statement. Statements after **when** will be executed only if ***expr*** evaluates to the same as the argument of the **case** statement, and it is the first ***expr*** to match from the list of **whens**.

Use

See **case** for details.

See also

case, otherwise, endcase

while

Syntax

while ***expr***

...

endwhile

Purpose

To execute a **while** loop.

Arguments

expr is the test expression for the loop.

Use

This command allows a sequence of statements to be executed while a condition is true. The expression ***expr*** is evaluated at the start of the loop before the contents of the loop are executed.

If the expression evaluates to 0 (false), S-Base jumps to the statement following the **endwhile** and takes no further action. Otherwise S-Base executes the statements within the loop repeatedly, until ***expr*** evaluates to 0 at the start of the loop.

Example

```
| find the first plant with a given row number
def proc find (row)
local i=1
while i < @recordcount plant
  load plant, i
  if plant.row = row then return i
  i += 1
endwhile
return -1
```

```
endproc
```

See also

endwhile, repeat

zap

Syntax

zap *filespec* [, *indexspec*]

Purpose

To initialise a data file or index.

Arguments

filespec is the file to initialise (if *indexspec* is not given or is –1).

indexspec is the index to initialise. If not given (or if it is –1), then the file and all indexes are initialised.

Use

This command allows you to remove all records from a file. By giving only the *filespec* parameter, the file is cleared to zero records and all indexes are destroyed.

You can destroy just an index by giving the *indexspec* parameter as well. This is not often useful, since **reindex** will rebuild the index for you.

zap has the effect of clearing the file *filespec*'s memory buffer to default values (spaces and zeros). See **zero** for an explanation of this.

Example

```
{ zap_file() checks with the user before zapping a file
(f must be a string holding the file's label). Returns 1
if the file has been zapped, else 0 }

def proc zap_file (f)

   | check if we are allowed to zap the file
   if proc check_if_ok ("About to zap file '" + f + "'")
      zap f
      return 1
   endif
   return 0
enddef
```

See also

zero

zero

Syntax

zero *filespec*

Purpose

To clear a file's memory buffer to default values.

Arguments

filespec specifies the file whose memory buffer is to be cleared.

Use

This command is useful for blanking the record buffer before the user is to enter a new record. It sets all string fields to spaces, and all other fields to zero.

An S-Base convention is that record 0 in each file holds a copy of the default record for that file. However, using **zero** is generally faster than loading record 0.

See also

zap

4. Function reference

@abs

Syntax

@abs ***number***

Purpose

To calculate the absolute value of a number.

Arguments

number is an integer or float number.

Returns

Integer or float – same type as ***number***.

@abs returns the absolute value of its argument.

Use

@abs allows you to remove any negative sign from a number.

Example

```
s:print @abs 4
4
s:print @abs -5.67
5.67
s:print @abs 0
0
```

@acos

Syntax

@acos ***number***

Purpose

To calculate the arc cosine of a number.

Arguments

number is the integer/floating number to use in the calculation in the range −1 to 1.

Returns

Float.

@acos returns the arc cosine of ***number*** in radians.

@addr

Syntax

(1) **@addr** ***variable*** | ***filefield***

(2) **@addr (buffer *handle*)**

(3) **@addr (def)**

(4) **@addr (bit *handle*)**

(5) **@addr (poll)**

Purpose

(1) To find the address of a variable or file field's value.

(2) To find the current address of the contents of a memory buffer.

(3) To find the address of the current application.

(4) To find the current address of the contents of a bitstring.

(5) To find the address of the last event returned from the Window Manager.

Arguments

variable is a variable of any type.

filefield is a file field of any type.

handle is a buffer or bitstring handle.

Returns

Integer.

@addr returns the appropriate address depending on its argument.

Use

The first form of **@addr** is used in situations where the programmer needs to access the data inside a variable or file field directly, or where the address must be passed to RISC OS using the **sys** command. **@addr** can be used on any variable or file field type. For arrays you can give a subscript specifying the element you want the address of.

Variable values always start on a 4-byte word boundary (eg address 4, 8, 12, 8192 etc. but not 493). However, values are stored without gaps after this: in an array of 4-byte integers all elements will be word aligned, but an array of 1-byte integers will be stored in successive bytes.

Note that the addresses returned from this function may change with time. Simple variables tend to stay in one place but large arrays may be moved by S-Base. You should use **@addr** immediately before you require the address. If you assign the result to a variable, note that even *this* may cause an array to move.

The second form of **@addr** allows to get the address of a buffer. This is the address of the first byte in the buffer. Since buffers are stored in shifting store you must not rely on this address staying constant. You

should obtain the address and use it immediately. You cannot pass the address to a command which may alter memory. For example, executing a **dim** command may alter the position of the buffer. Even calling a procedure may do the same.

The third form of **@addr** allows access to the current application definition. This is not documented here, and is intended only to access information not available through normal S-Base commands and functions, should this become necessary in future.

The fourth form of **@addr** allows you to get the address of a bitstring. The same notes regarding the second form apply here.

The fifth form of **@addr** allows access to the last event returned to S-Base from the Window Manager. This is useful in raw event handlers, since it allows the application to access the event.

Example

```
def proc load_sprites (filename, spritearea)
| Load in a sprite file
sys "OS_SpriteOp", 0, @addr filename, @addr spritearea
enddef
```

@adval

Syntax

@adval ***number***

Purpose

To check the status of one of the RISC OS buffers.

Arguments

number is a negative integer specifying the buffer to check.

Returns

Integer.

@adval returns a number depending on the buffer status. See the RISC OS Programmers' Reference Manual for a list of buffer numbers.

@altered

Syntax

@altered [(***handle***)]

Purpose

To determine if a card or table has been altered.

Arguments

handle is the handle of the object to check. If this is not supplied the

current object is used.

Returns

Integer.

@altered returns 1 if the object has been altered.

Use

This function returns true (1) if the object in question has been altered by the user since the start of the last poll command (eg **card poll**). If the user has not altered any element of the object, this function returns false (0).

Internally, the 'altered' flag is set to zero when a **card poll**, etc. command is received, and set to one when the user presses a key which alters a field. Cursor movement and clicking in fields will not cause this flag to become set.

@altered is generally used after a **card poll**, etc., to find out whether the user has changed anything, and therefore whether the data should be saved to disc or ignored.

Example

```
load title,recnum
card new c,"edit video"
card show c
card poll c
if @altered (c)
   update title,recnum
endif
```

See also

@justaltered

@asc

Syntax

@asc ***string***

Purpose

To find the ASCII code of a character.

Arguments

The first character of ***string*** is the character to convert.

Returns

Integer.

@asc returns the ASCII code of the first character of ***string***.

Use

See Appendix A for a list of ASCII codes. A string can only hold codes 1

to 255.

Example

```
s:print @asc "A"
65
```

See also

@chr

@asin

Syntax

@asin ***number***

Purpose

To calculate the arc sine of a number.

Arguments

number is the integer/float number to use in the calculation in the range −1 to 1.

Returns

Float.

@asin returns the arc sine of ***number*** in radians.

@atan

Syntax

@atan ***number***

Purpose

To calculate the arc tangent of a number.

Arguments

number is the integer/float number to use in the calculation.

Returns

Float.

@atan returns the arc tangent of ***number*** in radians.

@atanq

Syntax

@atanq (x, y)

Purpose

To calculate the arc tangent of a x/y.

Arguments

x is the numerator to use in the calculation.

y is the denominator to use in the calculation.

Returns

Float.

@atanq returns the arc tangent of x / y in radians, using the arguments to determine the sign of the result.

@bget

Syntax

@bget ***handle***

Purpose

To read a byte from an open file.

Arguments

handle is the handle of the file to read from.

Returns

Integer.

@bget returns the byte read from the file, or **-1** if the end of the file has been reached.

Example

This procedure prints the contents of a file as a list of decimal integers:

```
def proc dump (fname)
local fd = @openin fname

while !@eof fd
   ! read a byte and print it out
   print @bget fd:-4;
endwhile
enddef
```

See also

archive, bput, import, @eof, @openin, @openup

@bitfirstset

Syntax

@bitfirstset ***handle***

Purpose

To find out the first set bit in a bitstring.

Arguments

handle is the handle of the bitstring to test.

Returns

Integer.

@bitfirstset returns the index (from 0) of the first set bit in the bitstring, or −1 if there are no set bits.

Use

@bitfirstset is used in conjunction with selectors, which have a bitstring of selected entries. This function can return the number of the first entry the user has selected.

Example

This example prints out the text from the first selected entry in a selector entry:

```
| in a card c's click handler...'list' is a selector
s = @elementhandle (c, "list")
sel = @selectedhandle s
if @bitfirstset sel = -1
   print "There are no selected entries."
else
   print "The first selected entry is ";
   print @entrytext (s, @bitfirstset sel)
endif
```

See also

@bitsetcount, @bittest, @protectedhandle, @selectedhandle

@bitsetcount

Syntax

@bitsetcount ***handle***

Purpose

To find out how many bits of a bitstring are set (equal to 1).

Arguments

handle is the handle of the bitstring to test.

Returns

Integer.

@bitsetcount returns the number of set bits in the bitstring.

Use

@bitsetcount is used in conjunction with selectors, which have a bitstring of selected items. This function can return the number of items the user has selected.

Example

```
| in a card c's click handler...'list' is a selector
s = @elementhandle (c, "list")
sel = @selectedhandle s
```

```
print "You have selected ",@bitsetcount sel, "items from
this selector."
```

See also

@bitfirstset, @bittest, @protectedhandle, @selectedhandle

@bittest

Syntax

@bittest (*handle*, *bit*)

Purpose

To test a single bit of a bitstring.

Arguments

handle is the handle of the bitstring to test.

bit is the bit to test.

Returns

Integer.

@bittest returns 0 if the bit is clear, or 1 if the bit is set.

Use

@bittest is particularly useful with selectors, since these have a bitstring which describes which items are selected. This function can be used to determine whether a particular item is selected.

Example

```
{ 'sel' is the bitstring for a selector. We want to
print out the numbers of all the selected items }
local i

ffor i=0 to num_items - 1
   if @bittest (sel, i) then print i, " is selected"
next
```

See also

@bitfirstset, @bitsetcount, @protectedhandle, @selectedhandle

@bof

Syntax

@bof (*file* [, *index*])

Purpose

To find the last record in a file according to the given index ('bottom of file').

Arguments

file is the file to search.

index is the index to use for the search (given by index label or number). If none is given, record number order is assumed.

Returns

Integer.

@bof returns the record number of the last record in the file, or 0 if the file is empty.

Use

When no index is given, **@bof** will only ever return the last non-deleted record in the file (**@record count** *file*), or 0 if the file is empty.

See also

@tof, **@next**, **@previous**

@checkmem

Syntax

@checkmem *value*

Purpose

To check whether there is enough memory to perform an operation.

Arguments

value specifies the minimum amount of memory required to complete the operation.

Returns

Integer.

@checkmem returns 1 if at least *value* bytes of memory are available for use by the running program. If not, a value of 0 is returned.

Use

@checkmem should be used before a memory-intensive operation is performed to ensure the operation is possible. In general, an S-Base program should refuse to run if less than 50k of memory is available. Programs declaring big arrays should first check that enough memory is available for the array (+50k) to ensure that the program does not crash with an 'out of memory' error.

Example

```
| check if there is enough memory
if @checkmem (50000 + 160000)
   dim i reclist [40000]
```

```
   | plenty of memory, so do it
   proc process (reclist)
else
   error 1, "Not enough memory available"
endif
```

See also

@freemem

@chr

Syntax

@chr ***integer***

Purpose

To find the character corresponding to an ASCII code.

Arguments

integer is the ASCII code to look up.

Returns

String.

@chr returns a string containing the character for the ASCII code ***integer***.

Use

See Appendix A for a list of ASCII codes. ***integer*** should be in the range 1 to 255.

Example

```
s:print @chr 97
a
```

See also

@asc

@clickelement

Syntax

@clickelement [(***handle*** [, ***x*** , ***y***])]

Purpose

To find which element of a card or table the user has clicked on.

Arguments

handle is the handle of the card to check. If this is not supplied the current card is used.

x and ***y*** are the OS screen coordinates of the mouse click which has just occurred. If not given, the last mouse click is used.

Returns

Integer.

@clickelement returns the element found.

Use

With no coordinates given, this function returns the element of a card or table which was last clicked on. If there have been no mouse clicks on the card the value returned is undefined.

If coordinates are given, S-Base returns the top-most element under the coordinates, or –1 if there is no such element. The coordinates should be given in OS coordinates relative to the bottom left corner of the screen (not the top left of the window). This form of the function is useful in raw event handlers.

@clickelement is useful in click handlers, although **@clickelementlabel** is generally more useful.

See also

@clickelementlabel

@clickelementlabel

Syntax

@clickelementlabel [(***handle*** [, *x* , *y*])]

Purpose

To find the label of the current element of a card or table.

Arguments

handle is the handle of the card to check. If this is not supplied the current card is used.

x and *y* are the OS screen coordinates of the mouse click which has just occurred. If not given, the last mouse click is used.

Returns

String.

@clickelementlabel returns the label of the element found.

Use

With no coordinates given, this function returns the label of the element of the card or table last clicked on. If there have been no mouse clicks on the card, **@clickelementlabel** returns an empty string `""`.

If coordinates are given, S-Base returns the label of the top-most element under the coordinates, or `""` if there is no such element. The coordinates should be given in OS coordinates relative to the bottom left corner of the screen (not the top left of the window). This form of the function is useful

in raw event handlers.

@clickelementlabel is useful in click handlers to determine which format element the user has clicked on.

Example

```
| click handler for browser card
case @clickelementlabel
when "quit"
   card exit poll @handle
when "clear"
   zero curfile
   card update @handle
endcase
```

See also

@clickelement

@clickentry

Syntax

@clickentry [(***handle*** [, ***x*** , ***y***])]

Purpose

To find which entry of a table the user has clicked on.

Arguments

handle is the handle of the table to check. If this is not supplied the current table is used.

x and ***y*** are the OS screen coordinates of the mouse click which has just occurred. If not given, the last mouse click is used.

Returns

Integer.

@clickentry returns the entry found.

Use

With no coordinates given, this function returns the entry of the table which was last clicked on. If there have been no mouse clicks on the table the value returned is undefined.

If coordinates are given, S-Base returns the entry under the coordinates, or −1 if there is no such entry (e.g. the coordinates are beyond the end of the table). The coordinates should be given in OS coordinates relative to the bottom left corner of the screen (not the top left of the window). This form of the function is useful in raw event handlers.

See also

@clickelement

@clickmouse

Syntax

@clickmouse

Purpose

To find out which mouse button was pressed when the user last clicked the mouse.

Returns

Integer.

@clickmouse returns a code for the mouse buttons pressed.

Use

@clickmouse is useful in click handlers to find out which mouse button the user pressed.

The number returned is a value directly from the Window Manager made up of the following elements:

Number	Meaning
1	Click or double click with Adjust.
2	Click with Menu (all Menu clicks return this).
4	Click or double click with Select.
16	Drag with Adjust.
64	Drag with Select.
256	Click with Adjust.
1024	Click with Select.

The codes returned for any particular icon button type depend on the icon button type (the codes above apply for button type 10). See the RISC OS Programmers' Reference Manual for more details.

Example

```
| click handler for up arrow button

{ increment the value if Select is pressed, decrement
for Adjust }
case @clickmouse
when 1024
```

```
   value += 1
when 256
   value -= 1
endif

| update the card
card update @handle, "value"
```

See also

@clickelement

@cos

Syntax

@cos ***number***

Purpose

To calculate the cosine of a number.

Arguments

number is the integer/float number in radians to use in the calculation.

Returns

Float.

@cos returns the cosine of ***number***.

@cosh

Syntax

@cosh ***number***

Purpose

To calculate the hyperbolic cosine of a number.

Arguments

number is the integer/float number in radians to use in the calculation.

Returns

Float.

@cosh returns the hyperbolic cosine of ***number***.

@cursor

Syntax

@cursor [(***handle***)]

Purpose

To find the cursor position within an element of a card or table.

Arguments

handle is the handle of the card to check. If this is not supplied the current card or table is used.

Returns

Integer.

@cursor returns the cursor position, numbered from 0.

Use

This function returns the position of the cursor within the current field of a card or table.

This function is sometimes useful in keypress handlers. Note that since the keypress handler is called after the key has been processed, the value of the field may have been updated. However, S-Base will not have altered the cursor position or moved the cursor yet. Nor will it have called any exit handlers, etc. This is done after the keypress handler has finished, if necessary.

See also

@element, @handle

@date

Syntax

@date

Purpose

To get the current date.

Arguments

None.

Returns

Date.

@date returns the current date, read from the system clock.

@datesi

Syntax

@datesi ***string***

Purpose

To convert a string representation of a date interval to a date.

Arguments

string is the string to convert.

Returns

Date.

@datesi returns the date interval represented by the string.

Example

```
s:print @date
01-01-92
s:print @date + @datesi "1d"
02-01-92
s:print @date + @datesi "1y"
01-01-93
```

See also

@date, @datesn

@datesn

Syntax

@datesn ***string***

Purpose

To convert a string in natural format to a date.

Arguments

string is the string to convert.

Returns

Date.

@datesn returns the date representation of ***string***.

Use

@datesn is useful for converting user input to internal values. S-Base uses this function to process dates entered by the user.

For parts of the date not specified, S-Base uses the current date.

Example

```
| Assuming today is 18 May 1992:
s:print @datesn "30/12/89"
30-12-89
s:print @datesn "12"
12-05-92
s:print @datesn "12/6"
12-06-92
```

See also

@date, @datesi

@deg

Syntax

@deg ***number***

Purpose

To convert from radians to degrees.

Arguments

number is the number to use in the calculation.

Returns

Float.

@deg returns ***number*** radians expressed in degrees.

Example

```
print @deg @pi
180.00
```

See also

@rad

@deleted

Syntax

@deleted (***filespec***, ***record***)

Purpose

To determine whether a record is marked as deleted.

Arguments

filespec specifies the file containing the record.

record specifies the record to check.

Returns

Integer.

@deleted returns non-zero if the specified record is marked as deleted, otherwise 0.

Use

@deleted is useful when manually processing a file without using **select** or **@find**. You can choose to ignore deleted records by checking the value of this function before loading and processing a record.

See also

delete, **pack**

@dollar

Syntax

@dollar ***number***

Purpose

To truncate a number to a currency value (2 decimal places).

Arguments

number is the number to truncate.

Returns

Float.

@dollar returns the truncated number.

Use

@dollar is useful when you use a float to store currency values. For example, with British currency the fractional part stores the number of pence. Before truncating the value, S-Base adds 0.0001, to ensure that values very close to 1p round up.

Example

```
s:print @dollar 2
2.00
s:print @dollar 2.705:1:5
2.70000
s:print @dollar 2.709
2.70
s:print @dollar 2.7099
2.71
```

See also

@round

@elaltered

Syntax

@elaltered [(***handle***)]

Purpose

To determine if an element of a card or table has been altered.

Arguments

handle is the handle of the object to check. If this is not supplied the current object is used.

Returns

Integer.

@elaltered returns true (non-zero) if the current element (containing the cursor) of the object has been altered since the cursor entered the element.

Use

Internally, this flag is set to zero when the cursor enters a new field, and set to one when the user presses a key which alters the field. Cursor movement and clicking in fields will not cause this flag to become set.

@elaltered is sometimes used in keypress handlers, possibly to protect or unprotect elements of the format in response to user action. It checks whether the field has been altered.

See also

@altered, @justaltered, @entryaltered

@element

Syntax

@element [(***handle***)]

Purpose

To find the current element of a card or table (the one containing the cursor).

Arguments

handle is the handle of the object to check. If this is not supplied the current card or table is used.

Returns

Integer.

@element returns the current element number.

Use

This function returns the *current element* of a card or table — the element containing the cursor. If no element contains the cursor, this function returns –1.

This function is useful as an alternative to **@elementlabel** when there is a relationship between the physical position of one group of fields and another which they affect. For example, entering a number into a series of VAT-exclusive fields may cause a corresponding series of VAT-inclusive fields to change, and vice versa. If the fields are laid out so that each VAT-exclusive element precedes the corresponding VAT-inclusive element, it is easy to work out which field to update when another is altered. This can reduce the size of **case** statements in a keypress handler, but does require that the elements are not re-ordered in the format list.

Example

The following example is for two keypress handlers for a set of VAT-exclusive and VAT-inclusive fields in a format. When the user enters a value in the VAT-exclusive field, the corresponding VAT-inclusive field must be updated, and vice versa. The expression in the exclusive fields is `video.price[...]`. The expression in the inclusive fields is `video.price[...] * vat_rate`. Each inclusive field is the next element after the corresponding exclusive element when the format's

element list is viewed in physical order.

```
| handler for exclusive fields
| update the inclusive field
if @justaltered then card update @handle, @element + 1


| handler for inclusive fields
| update the exclusive field
if @justaltered
   | alter the exclusive price
   video.price[@element // 2] = @elementvalue @element /
1.175
   | and update the exclusive field
   card update @handle, @element - 1
endif
```

See also

@elementlabel

@elementhandle

Syntax

@elementhandle (***handle***, ***elementspec***)

Purpose

To get the object handle of an object attached to an element in a card.

Arguments

handle is the handle of the object to check.

elementspec is the element to which the object to search for is attached.

Returns

Integer.

@elementhandle returns the handle of the object attached to an element in another object.

Use

@elementhandle is often used with selectors. A **card new** command will create a new selector for each selector element in the card, but will not tell you the handles of these selectors.

By using this function you can get the handle of any selector in a card, which allows you to add items to the selector, etc.

Example

```
{ this example draws a pupil card on the screen with a
selector containing all pupil names }
```

```
| create the card & selector
card new c,"pupil cards"

| get the selector's handle
s = @elementhandle (c, "list")

{ get the handle of the 'selected' and 'protected'
bitstrings }
sel = @selectedhandle s
prot = @protectedhandle s

| store all the pupil names in the selector
t = @recordcount pupil

ffor i=1 to t
   load pupil,i
   selector insert s,@str (i)+ " "+@trim pupil.surname
next

| display the card & let the user play
card show c
card poll c
card dispose c

enddef
```

See also

@protectedhandle, @selectedhandle

@elementlabel

Syntax

@elementlabel [(*handle*)]

Purpose

To find the label of the current element of a card or table.

Arguments

handle is the handle of the object to check. If this is not supplied the current card or table is used.

Returns

String.

@elementlabel returns the label of the current element.

Use

This function returns the label of the *current element* of a card or table — the element containing the cursor. If no element contains the cursor, this function returns an empty string `""`.

This function is useful in keypress handlers, etc. It allows a handler to operate differently depending on which element contains the cursor.

Example

```
| keypress handler for several elements
case @elementlabel
when "title"
   ...handle keypress in 'title' field
when "rating"
   ...handle keypress in 'rating' field
endcase
```

See also

@element

@elementvalue

Syntax

@elementvalue (***handle***, ***elementspec***)

Purpose

To determine the value stored in a card element's text buffer.

Arguments

handle is the handle of the card containing the element.

elementspec specifies the card element to scan. This may be the element's label or number.

Returns

Type varies.

@elementvalue returns the value of the specified field.

Use

@elementvalue converts text entered by the user in a field into a value. The type of the value produced depends on the type of the field, which is determined by the type of expression which was evaluated to create the field in the first place. This function converts the text into a value of the same type as the field.

@elementvalue is useful within key handlers to determine the value the user has entered into a field which doesn't have *auto-recalc*. Fields which contain expressions will not work with *auto-recalc*, because S-Base does

not know where to put the field's value. A user keypress or exit field handler must insert the value manually.

Example

Assuming a field containing the VAT-inclusive price:

```
title.price * vat_rate
```

The following keypress handler will assign the correct value into `title.price` and update the value bar.

```
| handler for 'price' field of title card
title.price = @elementvalue (@handle, @element) /
vat_rate
card update value @handle
```

@entry

Syntax

@entry [(***handle***)]

Purpose

To find the current entry of a report or table.

Arguments

handle is the handle of the object to check. If this is not supplied the current report or table is used.

Returns

Integer.

@entry returns the entry number.

Use

This function returns the current entry number for an object.

For reports, it returns the entry about to be added to a report. It is used in handlers associated with reports, particularly the area entry handler. Entries are numbered sequentially from the start of the first page to the end of the last page.

For tables, **@entry** returns the entry containing the cursor. Within a *request entry* handler it returns the entry being requested by S-Base.

@entry can be used to provide the record number which should be loaded for the current report entry. If the report is more complex than simply a record list from a file, then **@entry** can be compared with **@lastentry**. If these differ, S-Base is asking for a new entry. Otherwise S-Base is repeating a request for the previous entry.

In general, **@entry** will increment with each call to the report's area entry

handler. However, after S-Base has generated a new entry it checks to see if it has overflowed the current column. If so, the entry is removed and started in the next column (possibly on the next page). This can cause two or more calls to a report's area entry handler with **@entry** having the same value.

For tables, entries can be requested in any order. Entries are always numbered from zero.

For reports, entries start at zero, and are guaranteed to be requested sequentially. S-Base will not skip entry numbers as it generates reports.

Example

```
| area entry handler for a video catalogue report
| just load the current video record

recnum = @entry + 1
| we add 1 since entries are numbered from zero
if recnum <= @recordcount video
   load video, @entry + 1
else
   report exit run @handle
endif
```

See also

report new, table new, @lastentry

@entryaltered

Syntax

@entryaltered [(***handle***)]

Purpose

To determine if an entry of a table has been altered.

Arguments

handle is the handle of the table to check. If this is not supplied the current object is used.

Returns

Integer.

@entryaltered returns true (non-zero) if the current entry (containing the cursor) of the table has been altered since the cursor entered the entry.

Use

Internally, this flag is set to zero when the cursor enters a new entry, and set to one when the user presses a key which alters any field in the entry.

Cursor movement and clicking in fields will not cause this flag to become set.

@entryaltered is generally used in exit entry handlers to decide whether the entry needs to be saved. If the entry has not been altered there is no need to waste time saving it.

See also

@altered, @justaltered, @elaltered

@entrycount

Syntax

@entrycount ***handle***

Purpose

To find out the number of entries in a table or selector, etc.

Arguments

handle is the handle of the object to check.

Returns

Integer.

@entrycount returns the number of entries in the object.

Use

@entrycount is useful when you want to know how many entries are currently in an object. Objects which do not have entries (for example cards) return -1.

See also

@entrytext

@entrytext

Syntax

@entrytext (***handle***, ***entry***)

Purpose

To obtain the text in an entry of a selector, menu, etc.

Arguments

handle is the handle of the object to check.

entry is the entry (numbered from 0) to check.

Returns

String.

@entrytext returns a string containing the entry text.

Use

@entrytext is useful in menu select handlers for decoding the user's menu selection. The function **@menuselect** provides you with the item number the user selected. This function can convert that item number to a string.

Note that for menus only the item string itself (plus any keyboard short-cut) is returned. None of the special characters mentioned in **menu insert** will be included in the returned string. In other words, a ticked item will not be preceded by an exclamation mark.

@entrytext is also useful for selectors for converting an entry number to a text string.

Example

Using the menu shown in **menu insert**:

```
s:print @entrytext (m, 0)
Info
s:print @entrytext (m, 2)
Funny ~!,:xx item
s:print @entrytext (m, 3)
Keyboard ^F1
s:print @entrytext (m, 5)
fadetick
```

@eof

Syntax

@eof ***handle***

Purpose

To determine whether the end of a file has been reached.

Arguments

handle is the handle of the file to check (returned from, e.g., **@openin**).

Returns

Integer.

@eof returns 1 if the end of a file has been reached, else 0.

Use

@eof is used when processing a file, to determine when to stop. S-Base will generate an error if you attempt to read past the end of a file.

Example

```
| open the file with 32k buffer
f = @openin ("infile", 32)
| continue until end of file
while !@eof f
```

```
   proc process_line (f)
end_while
close f
```

@errnum

Syntax

@errnum

Purpose

To determine the most recent error.

Arguments

None.

Returns

Integer.

@errnum returns 0 if no error has occurred, otherwise it returns the error number.

Use

@errnum is used in error handlers to find out which error occurred. If the error state is set to two, then this function can be used to check for an error after a database operation, etc.

Example

```
error set 2
load title, 49
error set 1
if @errnum
   proc display_error ("database", @errstr)
end_if
```

See also

def local error, **@errstr**, **@eval**

@errstr

Syntax

@errstr

Purpose

To determine the most recent error message.

Arguments

None.

Returns

String.

@errstr returns an empty string if no error has occurred, else it returns the error message.

Use

@errstr returns the error message in error handlers to find out the last error message generated by S-Base.

Example

See **@errnum** for an example.

See also

@errnum, **@eval**

@eval

Syntax

@eval (***string***)

Purpose

To evaluate an expression.

Arguments

string contains the expression to evaluate.

Returns

Type depends on expression.

@eval returns the result obtained from evaluating the expression in ***string***. If an error occurs during evaluation, the value is undefined.

Use

@eval is useful for evaluating expressions typed in by the user, and for calling procedures named at run-time within S-Base. **@eval** can cause an error if asked to evaluate an invalid expression. It is common to set the error state to two before calling this function.

Example

```
s:a="5*6"
s:b="a+7"
s:a = @eval (a)
s:print a
30
s:print @eval (b)
37
s:print @eval (b + "*2")
44
```

To call a procedure whose name is stored in `proc_name`:

```
@eval ("proc " + proc_name)
```

See also

@teval

@exp

Syntax

@exp *number*

Purpose

To calculate the exponent of a number.

Arguments

number is the number to use in the calculation.

Returns

Float.

@exp returns e^{number} where e is the constant 2.718281828.

@ext

Syntax

(1-3) **@ext** ***handle***

(4) **@ext** ***filefield***

Purpose

(1) To find the size of a file.

(2) To find the size of a buffer.

(3) To find out in a table extension handler by how much the table is to be extended.

(4) To find the number of bytes stored in an external file field.

Arguments

handle is the handle of the object to check.

filefield is an external file field.

Returns

Integer.

(1) **@ext** returns the size of the file in bytes.

(2) **@ext** returns the size of the buffer in bytes.

(3) **@ext** returns the number of entries by which the table is to be extended (negative if the table is being contracted).

(4) **@ext** returns the number of bytes held in the external file field.

Use

@ext is used to find the 'size' of an object. If the object has no size (e.g. a card), −1 is returned.

The third form above relates to tables and this is only used within a table extension handler. This handler is called when a table wants to expand or contract. The handler must decide whether to allow this change in entry count, and can look at the value of **@ext** to decide whether or not it is valid.

The fourth form above allows an application to find the number of bytes taken up by a file field.

Example

```
s:f = @openout "file"
s:set ptr f,10
s:print @ext f
10
s:close f
```

See also

@externalhandle, @openin, @ptr, set ext, table set ext, table ext

@externalhandle

Syntax

@externalhandle ***filefield***

Purpose

To find the file handle of a file's disc heap.

Arguments

filefield is an external file field.

Returns

Integer.

@externalhandle returns the file handle of the disc heap which holds the value for ***filefield***.

Use

@externalhandle is useful for direct access to the disc heap. This is sometimes necessary in situations where an external application must be told to act on a file field. The file handle (from this function), file position (from **@ptr**) and the size of the file field (from **@ext**) can be obtained for this purpose.

Note that the file handler returned is a RISC OS file handle suitable for passing to other applications. You cannot use this handle with functions like **@ptr** and **@ext**.

See also

@ext, @ptr

@filefield

Syntax

@filefield (***file***,***field*** [***subscript***])

Purpose

To evaluate a file field reference by file and field numbers.

Arguments

file is the file containing the field.

field is the field within the file to access.

If the field is an array, there must be a ***subscript*** which lists the subscripts to access.

Returns

Same type as field.

@filefield returns the value of the referenced field.

Use

@filefield is used in general purpose routines which do not want to be file-specific. It is possible to write routines which are passed a file label or number and possibly a field label or number, and these routines can perform actions on any file which is passed.

Example

Assuming plant file with fields 'name', 'type' and 'price [5]'

```
s:plant.name = "S-Base"
s:plant.price [3] = 5.67
s:print @filefield (plant, "name")
S-Base
s:print @filefield ("plant", "price" [3])
5.67
```

See also

file set field

@find

Syntax

@find ([**update**]***filespec***, ***indexspec*** [, {***value***}$^{+}$])

Purpose

To find the first record matching given criteria.

Arguments

filespec is the file to search.

indexspec is the index to use for the search.

value is a value to use for the search.

Returns

Integer.

@find returns the record number of the found record, or 0 if no record was found matching the criteria.

Use

@find is used to search for records matching certain criteria. It will search an index for the first record which matches all the search fields given.

The search fields should be given in the same order as fields in the index, and the maximum number of search fields allowed is equal to the number of fields in the index.

String fields will be matched up to the length of the specified search string. Thus specifying an empty string will always match a string field.

Note that if no search fields are specified, **@find** will return the first record in the file (as given by the ordering of the specified index).

If **@find** cannot find any record matching the criteria, or if there are no records in the file, **@find** will return 0.

It is possible to specify ***indexspec*** as –1. This means to use the normal record number ordering of a file. No values may be specified in the search, so **@find** will always return the first non-deleted record in the file if there are any records, otherwise 0.

Updating

If **update** is given in the **@find** function, S-Base will update ***filespec***'s record buffer with the last record checked before the function completed. The record will contain the values of all the fields contained within the index. This feature does not operate over a network.

For example, take an index on a 'name' and 'age' field in an address file. Imagine **@find** is used to search for a name of 'Smith', with no age given, and it finds it. **@find** will return the record number of the first 'Smith' in the file. If **update** is given, S-Base will update the address record with the name and age of the record found. The full name stored in the record would be stored in `'address.name'` and the age in `'address.age'`.

Updating is often useful to speed up the processing of a file. If all the information you need to process is stored in an index, you need not load each record from the data file. Just specify **update** in the **@find** function and the data will be loaded quickly from the index.

Example

Assume a video file with an index called 'cat,name' (index 0) on the 'category' and 'name' fields.

```
@find (video, "cat,name")
{ returns the first record in the video file in order of
category, name }

@find (video, 0, 'A')
| returns the first record in category 'A'

@find (video, "cat,name", 'A', "Wrath")
{ returns the first video in category 'A' whose name
starts with 'Wrath' }
```

See also

select, @findn

@findelem

Syntax

@findelem (***field*** | ***array*** [***subscript***], [***count***,] ***value***)

Purpose

To find a value in an array variable or file field.

Arguments

array is the array to search. This can either be a file field (eg title.price) or a variable. It must be an array.

subscript is a normal subscript or subscripts (each enclosed in square brackets) to start the search from (start of array by default).

count is the number of elements to search.

value is the value to search for.

Returns

Integer.

@findelem returns the subscript where the value was first found, or –1 if no match was found.

Use

@findelem performs a rapid search of an array variable or file field. It is useful in a variety of situations. By specifying the start subscript it is possible to continue a search after the last element found.

All types of arrays can be searched, with the exception of arrays of structures/externals. The value given must match the type of the array.

Searches of file fields of type string array and of variable X arrays do not work as quickly as other types.

Note that if the ***count*** parameter is given, the search will terminate after ***count*** elements have been searched. This may cause **@findelem** to return −1 even though matches occur later in the array.

Multi-dimensional arrays can be searched. The ***subscript*** is given as usual, with all dimensions in square brackets. ***count*** is still a single integer, and still gives the number of *elements* to search. Thus you should be careful to multiply the count by any necessary subscript sizes. **@findelem** returns the subscript of the value found as a single integer. This is obtained by multiplying out the individual subscripts as discussed under the **dim** command.

Example

```
s:dim i a [10]
s:dim s b [10]
s:a [5] = 56
s:print @findelem (a, 56)
5
```

This next line fails because it starts the search at element 6:

```
s:print @findelem (a[6],56)
-1
```

This next line fails because it only searches the first five elements (subscripts 0-4):

```
s:print @findelem (a [0], 5, 56)
-1
s:b[0] = "S-Base"
s:b[6] = "S-Base"
s:print @findelem (b[0],"S-Base")
0
s:print @findelem (b[1],"S-Base")
6
s:print @findelem (b[7],"S-Base")
-1
```

An example of multi-dimensional arrays:

```
s:dim i test [10] [20]
s:test [1] [10] = 23
s:print @findelem (test [1] [5], 23)
30
```

@findn

Syntax

@findn ([**update**] ***filespec***, ***indexspec*** [, {***value***}+])

Purpose

To find the record which is the nearest for given criteria.

Arguments

filespec is the file to search.

indexspec is the index to use for the search.

value is a value to use for the search.

Returns

Integer.

@findn returns the record number of the found record, or 0 if no records are in the file.

Use

@findn is used to search for records matching certain criteria. It will search an index for the nearest match for the search fields given. If an exact match is available, **@findn** will return the same value as **@find**. otherwise **@findn** will return the record number of the record which best matches the criteria (in some situations, the second best match is returned).

The search fields should be given in the same order as fields in the index, and the maximum number of search fields allowed is equal to the number of fields in the index.

String fields will be matched up to the length of the specified search string. Thus specifying an empty string will always match a string field.

Note that if no search fields are specified, **@findn** will return the first record in the file (as given by the ordering of the specified index).

If there are no records in the file, **@findn** will return 0.

It is possible to specify ***indexspec*** as −1. This means to use the normal record number ordering of a file. No values may be specified in the search, so **@findn** will always return the first non-deleted record in the file if there are any records, otherwise 0.

The notes under **@find** regarding updating also apply to **@findn**.

Example

Assume a video file with an index called 'cat,name' (index 0) on the 'category' and 'name' fields.

```
@findn (video, "cat,name")
```

```
{ returns the first record in the video file in order of
category, name }

@findn (video, 0, 'A')
| returns the first record in category 'A'

@find (video, "cat,name", 'A', "Wrath")
{ returns the first video in category 'A' whose name
starts with 'Wrath' or is the nearest match (eg category
'B' name 'Abraham') }
```

See also

select, @find

@formstr

Syntax

@formstr (***string***, ***width***)

Purpose

To word-flow a string for display.

Arguments

string is the string to operate on.

width specifies the line width into which words from the string are to be formatted.

Returns

String.

@formstr returns the formatted string.

Use

@formstr is useful for generating output where words must flow from one line to the next without breaking. It converts ***string*** into a string containing a number of 'lines' each of ***width*** characters. If the string is written to a file one 'line' at a time, the result is word-wrapped text. Note that no newline characters are inserted into the string.

Example

This procedure prints out a string with a given number of characters per line, ensuring that words do not break.

```
def proc display (str, line_len)
local i

str = @formstr (str, line_len)
for i = 1 to @len str step line_len
```

```
    print @mid (str, i, line_len)
next
enddef
```

@fpeek

Syntax

(1) **@fpeek** ***address***

(2) **@fpeek** (***address***, ***size***)

Purpose

To read a float from a memory location.

Arguments

address is the memory address to read from. It need not be word aligned.

size is the size of the float value to read. This is either 4 or 8 (8 by default).

Returns

Float.

@fpeek returns the float value read from ***address***.

See also

@ipeek

@freemem

Syntax

@freemem

Purpose

To determine the amount of free memory available.

Arguments

None.

Returns

Integer.

@freemem returns the number of bytes available to S.

Use

@freemem is useful for keeping a check on available memory in the machine. It is accurate to the nearest 1024 bytes.

Note that S-Base may be able to free more memory than this function indicates if required. However, you should be wary of declaring arrays which use more memory than apparently available.

See also

@checkmem

@handle

Syntax

@handle

Purpose

To discover the object handle of the current object (the one which called this handler).

Returns

Integer.

@handle returns the object handle.

Use

This function is only useful within handlers. It returns the handle of the object which called the handler. For example, in a keypress handler for a card, this function gives the card's handle.

@handle is useful where one handler can be called by several different cards and the particular card handle is not known. It is also useful in *extension* handlers for tables, since the table handle has not been placed in any variable when the handler is called.

@indexcount

Syntax

@indexcount ***filespec***

Purpose

To determine the number of indexes in a file.

Arguments

filespec specifies the file to check.

Returns

Integer.

@indexcount returns the number of indexes in the file ***filespec***.

Use

@indexcount is sometimes useful for performing an action on all indexes, eg reindexing all indexes in a file.

Example

```
| reindex all indexes in the title file
local i
for i = 1 to @indexcount title
   reindex title, i
next
```

@inkey

Syntax

(1) **@inkey** ***delay***

(2) **@inkey** ***key_number***

(3) **@inkey** -256

Purpose

(1) To wait for a certain time interval for a keypress.

(2) To scan the keyboard to see if a particular key is being pressed.

(3) To determine the operating system version.

Arguments

delay is the delay in centiseconds to wait for a key before giving up.

key_number is the internal key to scan for. See the RISC OS Programmers' Reference Manual for a list of internal key numbers.

Use (1)

@inkey is used to wait for a certain time for a keypress. If the user presses a key within the given time delay (or there is already a keypress in the keyboard buffer), **@inkey** returns the ASCII code of the key. Otherwise −1 is returned.

This use of **@inkey** has little use in a Wimp environment since all keyboard input is handled by the Window Manager.

Use (2)

@inkey is used to scan the keyboard to see if a certain key is being pressed. **@inkey** returns −1 if the key is being pressed, else 0. Note that the ***key_number*** is obtained from the RISC OS internal key numbers using the *exclusive or* (~) operator. For example, the code for either shift key is listed as 0. So to scan for Shift, set ***key_number*** to `0 ~ -1` (which is −1).

Use (3)

@inkey returns a number identifying the OS-Base version. Under RISC OS 2.00, this returns &a1 or 161.

Example

```
| Check if the user is holding down shift
if @inkey -1 then print "You are holding down shift"

| wait for one second for a keypress
a = @inkey 100
if a = -1
   print "You took too long"
```

```
else
   print "You pressed ", a:/c
end_if
```

@instr

Syntax

@instr (***string***, ***search_string*** [, ***pos***])

Purpose

To find occurrences of a string in another string.

Arguments

string is the string to search.

search_string is the string to look for.

pos is the position in string to start the search, default 1.

Returns

Integer.

@instr returns the first position in ***string*** where the string ***search_string*** was found, or 0 if not found.

Use

@instr is useful for rapidly searching strings, perhaps to break a string into words. The search starts at ***pos***: **@instr** cannot return a string position before this.

If ***search_string*** cannot be found in ***string***, **@instr** returns 0.

Example

```
{ break a string into separate words and print them on
separate lines }
def proc break_words (s)
local upto=1,p
while upto <= @len s
   p = @instr (s, " ", upto)
   if !p then p = @len s
   print @mid (s, upto, p - upto)
   upto = p + 1
endwhile
enddef
```

@int

Syntax

@int ***value***

Purpose

To convert a value to an integer.

Arguments

value is the value to convert.

Returns

Integer.

@int returns ***number*** converted to an integer.

Use

@int can be used on any type except string. It converts the value to an integer. For floats the value is truncated towards zero – i.e. negative fractions become less negative, and positive less positive . For external file fields, **@int** returns the block handle of the block where the field's data is stored. This will be –1 if the field does not hold anything.

Example

```
s:print @int 6
6
s:print @int 6.7
6
s:print @int -5.6
-5
s:print @int -.6
0
```

See also

@todate, @totime

@ipeek

Syntax

(1) **@ipeek** ***address***
(2) **@ipeek** (***address, size***)

Purpose

To read an integer value from a memory location.

Arguments

address is the memory address to read from. It need not be word aligned.

size is the size of the integer value to read. This can be from 1 to 4 (4 by default).

Returns

Integer.

@ipeek returns the integer value read from ***address***.

See also

@fpeek

@justaltered

Syntax

@justaltered [(***handle***)]

Purpose

To determine if an element of a card or table has been altered.

Arguments

handle is the handle of the object to check. If this is not supplied the current object is used.

Returns

Integer.

@justaltered returns true (non-zero) if the current element (containing the cursor) of the object has been altered by the user's last keypress.

Use

Internally, this flag is set to zero when the cursor enters a new field, and set to one when the user presses a key which alters the field. Cursor movement and clicking in fields will not cause this flag to become set.

@justaltered is generally useful in keypress handlers, possibly to protect or unprotect elements of the format in response to user action. Another common use for this function is where the contents of a field must be searched on as the user types. It would be inefficient to search every time the user presses the left arrow key. The keypress handler could check that **@justaltered** returns true before doing the search.

Example

```
| keypress handler to search for what the user has
| entered and display it in another field
| video_name is the variable behind the field where
| the user enters the name. "found" is the element
| where we put the title we found in the file

| if the field has been altered, find nearest match
if @justaltered
   load video, @findn (title, 0, video_name)

   | display nearest match in 'found' field
   card update @handle, "found"
endif
```

See also

@altered, @justaltered, @elaltered, @entryaltered

@key

Syntax

@key [(***handle***)]

Purpose

To determine the last key pressed by the user.

Arguments

None.

Returns

Integer.

@key returns the key code of the last key pressed by the user.

Use

@key is used in keypress handlers to determine what key the user has just pressed. It should not be used outside keypress handlers. The key codes used by S-Base are those used by the Window Manager.

Example

```
! keypress handler for a card
case @key
when 27  { Escape }
   card exit @handle
endcase
```

@lastentry

Syntax

@lastentry [(***handle***)]

Purpose

To find the last entry that was requested by S-Base in a report.

Arguments

handle is the handle of the report to check. If this is not supplied the current report is used.

Returns

Integer.

@lastentry returns the last entry number generated by S.

Use

This function returns the entry number that S-Base last generated. It is used in handlers associated with reports, particularly the area entry

handler. Entries are numbered sequentially from the start of the first page to the end of the last page.

@lastentry can be compared with **@entry** to see if S-Base is asking for a new entry. Otherwise S-Base is repeating a request for the previous entry.

The first time S-Base requests an entry for a report, **@lastentry** returns −1.

Example

See **@entry** for an example.

See also

@entry, **report new**

@left

Syntax

@left (***string***, ***length***)

Purpose

To extract a left substring of a string.

Arguments

string is the string to process.

length is the number of characters to extract.

Returns

String.

@left returns a string consisting of the left-most ***length*** characters from ***string***.

Use

@left is useful in string processing. It will never return more than ***length*** characters, but may return less, if the length of ***string*** is less than ***length***.

Example

```
s:print @left ("Alfred", 2)
Al
s:print @left ("All of it", 50), "."
All of it.
s:print @len @left ("Nothing", 0)
0
```

See also

@right, **@mid**

@len

Syntax

@len ***string***

Purpose

To find the length of a string.

Arguments

string is the string to check.

Result

Integer.

@len returns the number of characters in ***string***.

Use

@len is useful in many situations, for example checking for a null string. It is better (and faster) to compare the length of a string with 0 than a string with `""`.

Example

```
s:print @len "A string  "
10
s:s = @trim "    "
s:print @len s = 0
1
```

@ln

Syntax

@ln ***number***

Purpose

To calculate the natural logarithm of a number.

Arguments

number is the integer/float number to use in the calculation.

Returns

Float.

@ln returns the logarithm to base *e* of ***number***, where *e* is 2.718281828.

Use

@ln is useful in mathematical calculations. To obtain a logarithm of ***x*** to base ***y***, use **ln** ***x*** / **ln** ***y***.

@loadindex

Syntax

@loadindex (***filespec***, ***index***, ***rec_array*** [, ***max_count***])

Purpose

To load ordered record numbers from an index.

Arguments

filespec is the file whose index is to be scanned.

index is the index to scan.

rec_array is an integer 4 array which will hold the record numbers.

max_count is the maximum number of record numbers to load (by default all are loaded).

Returns

Integer.

@loadindex returns the number of record numbers placed in ***rec_array***.

Use

@loadindex is useful when you need to quickly obtain all the record numbers in an index for ordering purposes. This list of record numbers can then be used with **@order** to order another array of record numbers.

At most ***max_count*** record numbers are placed into the array. Fewer will be placed if either the array is too small or the index does not contain enough record numbers.

@loadindex performs a fairly rapid search but the data is not cached, so repeated use of this function is discouraged.

If ***indexspec*** is –1, S-Base searches the file sequentially and returns a list of the record numbers of non-deleted records. This is generally not as fast as using an index.

See also

@order

@lock

Syntax

@lock [(***filespec*** [, ***recnum***])]

Purpose

To lock files and records from alteration by other users.

Arguments

filespec is the file to lock. If not given, all files are locked.

recnum is the record to lock. If not given, all records in the file are locked.

Returns

Integer.

@lock returns 1 if the lock is successful, 0 if the lock cannot be created because another user holds a lock on the same data.

Use

@lock is used to lock a record, an entire file, or all files in the application. In stand-alone mode this command always returns 1.

You should use **@lock** with no parameters during system-wide maintenance operations like a backup. You should lock an entire file before reindexing or packing it. Locking a single record should be done before loading the record for update.

If **@lock** fails your application should report an error to the user and cancel the operation.

Example

```
| procedure to allow editing of an address record
def proc edit_record (recnum)
local c

| lock the record
if @lock (address, recnum)

   | load it and let the user edit it
   load address, recnum
   card new c, "address"
   card show c
   card poll c
   if @altered c then update c, recnum
   card dispose c

   | & unlock it again
   unlock address, recnum
else
   | report an error
   error 1, "Record "+@str (recnum)+" is in used by
another user"
endif
enddef
```

See also

unlock

@log

Syntax

@log ***number***

Purpose

To calculate the logarithm to base ten of a number.

Arguments

number is the integer/float number to use in the calculation.

Returns

Float.

@log returns the logarithm to base 10 of ***number***.

See also

@exp, @ln

@match

Syntax

@match (***string, template***)

Purpose

To check a string against a wildcard template.

Arguments

string is the string to check.

template is the template to check the string against.

Returns

Integer.

@match returns 0 if the match failed or 1 if the match succeeded.

Use

@match checks the string against the template character by character, ensuring that each character of the string matches the template.

The template may contain *wildcard* characters:

Character	Matches
?	Match any character
*	Match any number of characters

@match is useful when you want to let the user enter a wildcard and then check it against a list of strings. Multiple *'s are handled correctly.

Example

```
s:print @match ("S-Base", "S-Base")
1
s:print @match ("Alfred", "Orange")
0
s:print @match ("crossword", "c?o*s*o*d")
```

```
1
s:print @match ("starts with", "sta*")
1
s:print @match ("end with this", "*this")
1
s:print @match ("if it contains this", "*contains*")
1
```

@max

Syntax

@max ({ *number* }+)

Purpose

To find the maximum of a set of values.

Arguments

number is an integer, float, date or time value.

Returns

Type varies according to arguments.

@max returns the maximum value from the list.

Example

```
s:print @max (1, 2, 3)
3
s:print @max (1, 7, 5.6, 3)
7.00
```

See also

@min

@mean

Syntax

@mean (***variable*** [, ***length***])

Purpose

To get the mean of all or part of an array.

Arguments

variable is a variable of array type to process. If a subscript is given, it is used as the starting element for the calculation.

length is the number of elements to use (all by default).

Returns

Float.

@mean returns the mean calculated from the array elements.

Use

@mean can be used with integer or float variables. It does a rapid calculation through the specified range (starting element and length) within the array.

Example

Given an array 'x' with 10 elements containing the values 0 to 9:

```
s:print @mean (x):1:6
4.500000
s:print @mean (x[5])
7.00
s:print @mean (x,5)
2.00
s:print @mean (x[7],1)
7.00
```

See also

@sdev, **@sum**

@menu

Syntax

@menu

Purpose

To find out the handle of the current menu.

Returns

Integer.

@menu returns the handle of the current menu, or −1 if there is none.

Use

@menu can only be used in menu select handlers. It can be useful for finding out which menu the user has selected from so that the menu can be altered. It is faster than using **@menufind** with the menu's label.

See also

@menufind

@menufind

Syntax

@menufind *label*

Purpose

To obtain the handle of a named menu.

Arguments

label is the label of the menu to locate.

Returns

Integer.

@menufind returns the handle of the menu, or –1 if not found.

Use

@menufind is most often used in menu maker handlers. It returns the handle of the given menu if it exists. S-Base will look for the menu in the menu resource and execute that resource if necessary, to create the menu. Once the menu is created this is not necessary.

This function returns –1 if the menu is not found. This should not normally happen — all menus used in menu maker handlers should exist in the menu resource.

To make S-Base forget the menu (because it is about to change), use **menu dispose**.

Example

See **menu new** for an example.

See also

menu new, **menu dispose**, **@menu**

@menuselect

Syntax

@menuselect *level*

Purpose

To find out which menu item the user clicked on.

Arguments

level is the menu level requested. Use 0 for the main menu, 1 for its submenu, etc.

Returns

Integer.

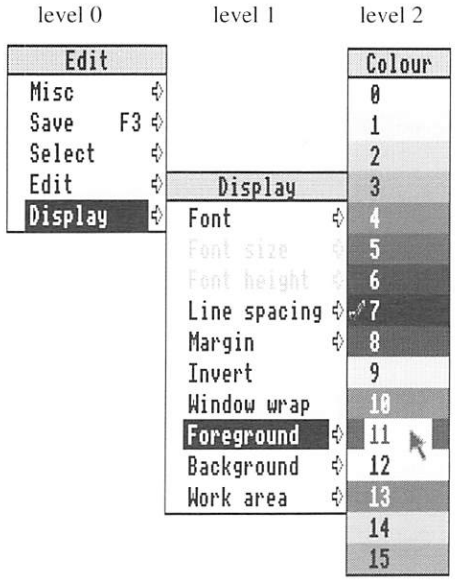
@menuselect returns the item (numbered from 0) selected from the given level of the current menu, or –1 if no item was selected.

Use

@menuselect is only used in menu select handlers. It allows you to determine which menu item the user has clicked on. This function will not operate outside a menu select handler, since there is no current menu.

For a simple menu with no submenus you should use level 0.

Example



From the menu above you can see that the user has selected Display–>Foreground–>11, so **@menuselect** will return the following:

```
s:print @menuselect 0
4
s:print @menuselect 1
7
s:print @menuselect 2
11
```

In this case using **@menuselect** with any other value (e.g. `@menuselect 29`) will return –1, since no selection was made at that level.

@mid

Syntax

@mid (*string*, *start_index* [, *length*])

Purpose

To extract a substring from a string.

Arguments

string is the source string.

start_index is the index of the first character to extract.

length is the number of characters to extract, by default the rest of the string.

Returns

String.

The extracted string.

Use

@mid is useful in string processing. It extracts ***length*** characters from string starting at ***start_index***.

If ***start_index*** is less than 1 or greater than the length of ***string***, **@mid** returns an empty string. **@mid** can never return a string longer than ***string***, and if ***length*** characters would extend past the end of ***string***, only as many characters as are available are extracted.

Example

```
s:print @mid ("The whole world", 6, 4)
hole
s:x = "S-Interpreter"
s:print @mid (x, @instr (x, "-") + 1)
Interpreter
s:print @mid ("Alfred", 3, 50), " was here."
fred was here.
```

See also

@left, **@right**

@min

Syntax

@min ({***number***}+)

Purpose

To find the minimum of a set of values.

Arguments

- ***number*** is an integer, float, date or time value.

Returns

Type varies according to arguments.

@min returns the minimum value from the list.

Example

```
s:print @min (1, 2, 3)
1
s:print @min (1, 2.0, 0.3)
0.30
```

@monotime

Syntax

@monotime

Purpose

To get the elapsed time in centiseconds.

Returns

Integer.

@monotime returns the current time in centiseconds (1/100 seconds).

Use

@monotime can be used to time operations, and to wait for a certain length of time. The value returned can only be considered relative to an earlier time as there is no point of reference.

Monotonic times are unsigned so they should be compared after subtraction as the example below shows.

Example

```
! wait for one second before continuing
wait = @monotime + 100
while wait - @monotime > 0
endwhile
```

@msg

Syntax

@msg *tag*

Purpose

To translate a tag string into a language string.

Arguments

tag is a string or an integer specifying the tag and the default language string for use if no match for the tag is found in the language file.

Returns

String.

@msg returns the translated string.

Use

@msg is useful when it is required that a program display prompts in a number of different natural languages. These prompts should be stored in a message file within the application directory called '`messages`', one per line, in the following format:

tag : translated string

If the tag specified in the **@msg** parameter cannot be found in the message file, **@msg** will return the default string specified after a colon (:) in ***tag***. If no default string is provided, **@msg** returns the tag given in ***tag***.

Example

Assuming the message file contains the following three lines:

```
nomem:Out of memory
help:Press <f1> for help
2:This is message 2
```

Then:

```
s:print @msg "nomem"
Out of memory
s:print @msg "45:Please enter the balance"
Please enter the balance
s:print @msg 2
This is message 2
s:print @msg "wow"
wow
s:print @msg "wow:bow wow"
bow wow
```

@mult

Syntax

@mult (***string***, ***length***)

Purpose

To ensure that the length of a string is a multiple of a given length.

Arguments

string is the string to process.

length specifies the length multiple for the string.

Returns

String.

@mult returns a string whose length is a multiple of ***length***, padding the string with spaces if necessary.

Use

@mult is useful for lining up items so a repeated **@pad** pattern can be appended.

Example

Without **@mult**, the dots below do not line up:

```
s:print @pad("S-Base",40," ."), 1
```

```
S-Base . . . . . .1
s:print @pad("An Introduction",40," ."), 2
An Introduction . . . . . 2
```

Using **@mult** they do:

```
s:print @pad(@mult("S-Base",2),40," ."), 1
S-Base . . . . . .1
s:print @pad(@mult("An Introduction",2),40," ."), 2
An Introduction  . . . . . .2
```

@next

Syntax

@next (*file*, *index*, *record*)

Purpose

To locate the next record in a file.

Arguments

file is the file to search.

index is the index (label or number) to use for the search. Use –1 for record number order.

record is the record to start the search from.

Returns

Integer.

@next returns the next record in the file after *record*, or –1 if *record* is the last one.

Use

@next is useful for browsing through a file one record at a time. It is often used to handle a 'next record' option in a record browser card.

You can use **@tof** to find the first record, then **@next** to move through all the records until it returns –1.

Example

```
{ print out all the records in the title file in order
of title }
local i

rec = @tof (title, "title")
while rec > 0
   load title, rec
   print rec:-4,title.title
   rec = @next (title, "title", rec)
```

```
endwhile
```

See also

@previous, @tof, @bof

@nextfree

Syntax

@nextfree *filespec*

Purpose

To find the next available record in a file.

Arguments

filespec gives the file to check.

Returns

Integer.

@nextfree returns the record number of the next available record in a file.

Use

@nextfree is used to determine which record number the next **create** is likely to use. This is often displayed in cards for the user's benefit. In network operation, **create** may have to use a different record because another user has got in first. Therefore **@nextfree** can only be used as a guide.

Example

```
| find the next free record
rec = @nextfree title
print "Expecting to use record ",rec
| save data to this record
create title
print "Actually used ",@record title
```

@onespace

Syntax

@onespace *string*

Purpose

To remove any superfluous spaces between words in a string.

Arguments

string is the string to convert.

Returns

String.

@onespace returns the converted string.

Use

@onespace is useful in conjunction with **@formstr** when formatting strings into a fixed width column. All sequences of more than one space within the string are converted to a single space.

Example

```
s:print @onespace ("This   is       a test.")
This is a test.
```

@openin

Syntax

@openin (***filename*** [, ***buffer_size***])

Purpose

To open a file for input.

Arguments

filename is the filename to open.

buffer_size is the size of the buffer to use (in kbytes, default 0).

Returns

Integer.

@openin returns an object handle if successful, or −1 if the open failed.

Use

This function is used to open a file for buffered input. S-Base allocates memory to the file buffer (if specified) and opens the file. A file may be opened for input as many times as required, although this is not generally useful. Note that the object handle returned is not the file handle, but a handle generated by S-Base.

S-Base will close files opened by this function automatically when execution is terminated.

Example

```
s:print @openin "notthere"
-1
```

Get the size of a file:

```
s:f = @openin "file"
s:print @ext f
56
```

See also

@openout, **@openup**, **close**

@openout

Syntax

@openout (*filename* [, *buffer_size*])

Purpose

To open a file for output.

Arguments

filename is the filename to open.

buffer_size is the size of the buffer to use (in kbytes, default 0)

Returns

Integer.

@openout returns a handle if successful, or −1 if the open failed.

Use

This function is used to open a file for buffered output. S-Base allocates memory to the file buffer (if specified) and opens the file. A file may not be opened for output twice (without closing the original handle). Note that the object handle returned is not the file handle, but a handle generated by S-Base.

Any existing file is erased in this operation. S-Base will close files opened by this function automatically when execution is terminated.

Example

```
s:f = @openout "file"
s:bput f,5
s:print @ptr f
1
```

See also

@openin, **@openup**, **close**

@openup

Syntax

@openup (*filename*)

Purpose

To open a file for output.

Arguments

filename is the filename to open.

Returns

Integer.

@openup returns a handle if successful, or −1 if the open failed.

Use

@openup is used to open a file for update. It is possible to read from and write to the file after it is opened by this function. The file is not affected by this function. A file may not be opened for update twice (without closing the original handle).

Note that the object handle returned is not the file handle, but a handle generated by S-Base.

S-Base will close files opened by this function automatically when execution is terminated.

Example

```
s:f = @openup "file"
s:bput f,42
s:set ptr f, 0
s:print @bget f
42
```

See also

@openin, @openout, close

@order

Syntax

@order (***order_array***, ***order_count***, ***rec_array***, ***rec_count***)

Purpose

To re-order an array of integers.

Arguments

order_array is an integer 4 array specifying the ordering to use.

order_count is the number of elements from ***order_array*** to use.

rec_array is the integer 4 array to be ordered.

rec_count is the number of elements from ***rec array*** to order.

Returns

Integer.

@order returns the number of records ordered.

Use

@order is useful for ordering record numbers according to an index. After doing a fast *sequential* search through the database for matching records, you can then order the records according to an index. Sequential searching is much faster than performing the search in index order (when the search is on non-indexed fields), since no non-sequential database reads are required.

The ordering should be assembled in ***order_array*** and the records to be ordered assembled in ***rec_array***. After the command has completed, all integers in ***rec_array*** will be in the same order as the integers in ***order_array***. ***rec_array*** must contain only integers present in ***order_array***, since no ordering is given for other integers. In the case where ***rec_array*** contains all integers from ***order_array***, the result will be that ***rec_array*** = ***order_array***.

Example

In this example there is a list of `nselection` record numbers from the title file called `selection`, which have been obtained by a sequential search. These record numbers must be ordered by title, using the `title` index.

```
def proc reorder (return selection, nselection)
local rec, count

{ get record numbers from this index into the order
array }
dim i rec [@recordcount title]

| load the record numbers into 'rec'
count = @loadindex (title, "title", rec)

{ order the records in selection[] according to this
index }
count = @order (rec, count, selection, nselection)

| & return the ordered selection
enddef
```

@pad

Syntax

@pad (***string***, ***width*** [, ***padding_string***])

Purpose

To pad or truncate a string to a certain width.

Arguments

string is the string to pad.

width is an integer specifying the width to pad the string to.

padding_string will be used to pad the string (a single space by default).

Returns

String.

@pad returns the padded string of length ***width***.

Use

@pad is used to ensure that a string is of a certain length. If ***string*** is too short, characters will be added at the end. If ***string*** is too long it will be truncated.

The padding string may be of any length and will be used as a repeating pattern to pad the string if it is not long enough.

Example

```
s:print @pad ("column1", 10, "."), "Now this"
column1...Now this
s:print @pad ("test", 10, "<pad>")
test<pad><
```

See also

@left, @mid, @mult, @onespace, @right

@page

Syntax

@page [(***report_handle***)]

Purpose

To determine the current page number while generating a report.

Arguments

report_handle gives the report handle for which the current page number is required.

Returns

Integer.

@page returns the current page number.

Use

@page is often used to display a page number at the top or bottom of reports. It is set to 1 when the report begins execution, and incremented each time a new page is started. If ***report_handle*** is not given, the current report is used — this is only allowed within a handler for the report.

Example

The following expression, placed in a format element of type prompt, in the 'Page footer' area, will place a page number at the bottom of each page:

```
"Page " + @str (@page)
```

@pi

Syntax

@pi

Purpose

To determine the value of PI.

Arguments

None.

Returns

Float.

@pi returns the value of PI (3.1415926536).

@point

Syntax

@point (*x*, *y*)

Purpose

To determine the colour of a pixel on the screen.

Arguments

x, *y* are the OS coordinates of the point to check.

Returns

Integer.

@point returns the colour of the pixel at *x*, *y* on the screen, or −1 if the point is off the screen.

@previous

Syntax

@previous (*file*, *index*, *record*)

Purpose

To locate the previous record in a file.

Arguments

file is the file to search.

index is the index (label or number) to use for the search. Use −1 for record number order.

record is the record to start the search from.

Returns

Integer.

@previous returns the previous record in the file before *record*, or −1 if *record* is the first one.

Use

@previous is useful for browsing through a file one record at a time. It is often used to handle a 'previous record' option in a record browser card.

You can use **@bof** to find the last record, then **@previous** to move through all the records until it returns –1.

Example

```
{ click handler for a 'previous record' button in a
browser card }
local rec

| get the previous record
rec = @previous (curfile, curindex, @record curfile)

| if there is one, load it and update the card
if rec > 0
   load curfile, rec
   card update c
   card set cursor c, 0
endif
```

See also

@next, **@tof**, **@bof**

@protectedhandle

Syntax

@protectedhandle ***handle***

Purpose

To find out the handle of the *protected* bitstring of a selector.

Arguments

handle is the handle of the selector to look at.

Returns

Integer.

@protectedhandle returns the handle of the bitstring found, or –1 if none was found.

Use

@protectedhandle is used to discover which items of a selector are protected from being clicked on by the user. The result is the handle of a bitstring which can be examined with the usual bitstring operations (**@bittest** in particular).

Example

See **@elementhandle** for an example.

See also

@bitfirstset, @bittest, @bitsetcount, @elementhandle, @selectedhandle

@ptr

Syntax

(1) **@ptr** ***handle***
(2) **@ptr** ***filefield***

Purpose

(1) To determine the current read or write position in a file.

(2) To determine the position of a file field's disc heap block in the disc heap.

Arguments

handle is the handle of the file to check (returned from **@openin**, for example).

filefield is an external file field.

Returns

Integer.

@ptr returns the value of the file pointer.

Use

@ptr can be used with external file fields to obtain the position of the data, perhaps for processing by another application.

Example

```
s:fd = @openout "test"
s:print @ptr fd
0
s:archive fd,56
s:print @ptr fd
4
s:set ptr fd,0
s:print @ptr fd
0
```

See also

@ext, @externalhandle

@rad

Syntax

@rad ***value***

Purpose

To convert a number from degrees to radians.

Arguments

value is the number to convert.

Returns

Float.

@rad returns ***value*** converted to radians.

Use

@rad is often used with **@sin** and **@cos**, etc, since these require their argument to be in radians.

Example

```
s:print @sin @rad 90
1.00
```

See also

@deg

@record

Syntax

@record ***filespec***

Purpose

To determine the last record accessed in a file (with **load**, **save**, etc.).

Arguments

filespec specifies the file to check.

Returns

Integer.

@record returns the last accessed record number for a file.

Use

@record is most useful for determining which record number **create** used to create a record. In stand-alone mode, **create** will always use **@nextfree**. In network mode it wil probably use **@nextfree**, but you cannot be sure — another station may issue a **create** command between your calls to **@nextfree** and **create**.

Example

```
create plant
```

```
print "Created record #", @record plant
```

@recordcount

Syntax

@recordcount *filespec*

Purpose

To find the number of records in a file.

Arguments

filespec gives the file for the record count.

Returns

Integer.

@recordcount returns the number of records in the file.

Use

@recordcount is useful where a file must be processed without using an index. Note that only the **create** command will increment the record count. **save** does not alter it.

Example

```
| print out all the video titles
for i=1 to @recordcount title
   load title, i
   print title.title
next
```

@recsize

Syntax

@recsize *file*

Purpose

To find the record size of a file.

Arguments

file is the file to check.

Returns

Integer.

@recsize returns the size of the record structure in bytes.

Use

This function may be useful for working out the rough size of a file on disc.

See also

@vrecsize

@right

Syntax

@right (***string***, ***length***)

Purpose

To extract a right substring of a string.

Arguments

string is the string to process.

length is the number of characters to extract.

Returns

String.

@right returns a string consisting of the right-most ***length*** characters from ***string***.

Use

@right is useful in string processing. It will never return more than ***length*** characters, but may return less, if the length of ***string*** is less than ***length***.

Example

```
s:print @right ("Alfred", 2)
ed
s:print @right ("All of it", 50), "."
All of it.
s:print @len @right ("Nothing", 0)
0
```

See also

@left, **@mid**

@rnd

Syntax

@rnd ***bound***

Purpose

To generate random integers.

Arguments

bound is an upper bound on the random number to be generated.

Returns

Integer.

@rnd returns a pseudo-random number in the range 0 to (***bound*** – 1) inclusive.

Example

```
s:print @rnd 10
6
s:print @rnd 2302
424
```

@round

Syntax

@round (***number***, ***decimals***)

Purpose

To round a float to a number of decimal places.

Arguments

number is the number to be rounded.

decimals specifies the number of decimal places to round the number to. If ***decimals*** is negative, the number is rounded to the nearest $10^{-decimals}$, i.e. $-decimals$ places to the left of the decimal point.

Returns

Float.

@round returns ***number*** rounded to ***decimals*** decimal places.

Example

```
s:print @round (123.456, 2):1:4
123.4600
s:print @round (123.456, -1):1:4
120.0000
s:print @round (123456, -1):1:4
123460.0000
```

@sdev

Syntax

@sdev (***variable*** [, ***length***])

Purpose

To get the standard deviation of all or part of an array.

Arguments

variable is a variable of array type to process. If a subscript is given, it is used as the starting element for the calculation.

length is the number of elements to use (all by default).

Returns

Float.

@sdev returns the standard deviation calculated from the array elements.

Use

@sdev can be used with integer or float variables. It does a rapid calculation through the specified range (starting element and length) within the array. It uses the $n-1$ rule for working out the deviation.

Example

Given an array 'x' with 10 elements containing the values 0 to 9:

```
s:print @sdev (x):1:6
3.027650
s:print @sdev (x[5])
1.58
s:print @sdev (x,5)
1.58
s:print @sdev (x[7],1)
0.00
```

See also

@mean, **@sum**

@selectedhandle

Syntax

@selectedhandle ***handle***

Purpose

To find out the handle of the *selected* bitstring of a selector.

Arguments

handle is the handle of the selector to look at.

Returns

Integer.

@selectedhandle returns the handle of the bitstring found, or −1 if none was found.

Use

@selectedhandle is useful for discovering which items of a selector have been selected by the user. The result is the handle of a bitstring which can be examined with the usual bitstring operations (**@bittest** in particular).

Example

See **@elementhandle** for an example.

See also

@bittest, **@bitsetcount**, **@elementhandle**, **@protectedhandle**

@sgn

Syntax

@sgn ***number***

Purpose

To calculate the sign of a number.

Arguments

number is an integer or float number.

Returns

Integer.

@sgn returns −1 if its argument is negative, 0 if it is zero and 1 if it is positive.

Example

```
s:print @sgn -56.3
-1
s:print @sgn 34
1
s:print @sgn 0
0
```

@sin

Syntax

@sin ***number***

Purpose

To calculate the sine of a number.

Arguments

number is the number in radians to use in the calculation.

Returns

Float.

@sin returns the sine of ***number***.

@sinh

Syntax

@sinh ***number***

Purpose

To calculate the hyperbolic sine of a number.

Arguments

number is the number in radians to use in the calculation.

Returns

Float.

@sinh returns the hyperbolic sine of ***number***.

@spc

Syntax

@spc ***count***

Purpose

To generate a string of spaces.

Arguments

count is the number of spaces to generate.

Returns

String.

@spc returns a string consisting of ***count*** spaces.

Example

```
local i
for i = 1 to 5
   print @spc i, i
next
```

Output:

```
 1
  2
   3
    4
     5
```

See also

@string

@sqr

Syntax

@sqr ***number***

Purpose

To calculate the square root of a number.

Arguments

number is the number to use in the calculation.

Returns

Float.

@sqr returns the square root of ***number***.

@str

Syntax

@str (*expression* [*fieldspec*])

Purpose

To create a string from formatted values.

Arguments

expression is an expression (of any data type) to be formatted.

fieldspec is a format specifier for the expression.

Returns

String.

@str returns the formatted expression.

Use

@str is used to convert data values to strings in the same way as **print**.

A ***fieldspec*** consists of up to three parts, all of which are optional. The full syntax is:

[***:/ subtype***] [**:** ***width*** [**:** ***decplaces*** | ***maxwidth***]]

See **print** for a discussion of this.

Example

```
@str (1:10)          "         1"
@str (1:-10)         "1         "
@str (1:10:2)        "      1.00"
@str (1.1:-10)       "1.10      "
@str (1.45:10:1)     "       1.5"
@str (1:/y:5)        "    Y"
@str (@time:/n)      "25/11/89"
@str (65:/c)         "A"
@str (2:/a:-5)       "C    "
@str (@time)         "12:56:04.25/11/89"
@str ("simon":10)    "     simon"
@str ("simon":-10)   "simon     "
@str ("simon":-3)    "simon"
@str ("alfred":1:3)  "alf"
```

@string

Syntax

@string (***count***, ***string***)

Purpose

To produce multiple copies of a string.

Arguments

count is the number of copies to produce.

string is the string to copy.

Returns

String.

@string returns a string consisting of ***count*** copies of ***string***.

Use

@string is sometimes useful for putting lines in text-only reports.

Example

The expression `@string (line_length, "=")` could be used in a text-only report format field to generate a line of equals signs in the report.

```
s:print @string(10,"-")
----------
s:print @string(5,"ab")
abababababab
s:print @len @string (0,"Nothing here")
0
```

See also

@spc

@stringize

Syntax

@stringize (***variable*** [, ***length***])

Purpose

To convert an integer array containing ASCII characters into a string.

Arguments

variable is an array containing the ASCII characters stored as integers. The subscript on this array specifies the first element of the array to convert (0 by default).

length is the number of characters to convert (all by default).

Returns

String.

@stringize returns the string.

Use

@stringize is useful for converting null-terminated strings returned from RISC OS and related modules into S-Base strings. The conversion starts at

the character specified by the subscript on the integer array, and continues until ***length*** characters have been converted, a null character (ASCII 0) is found or the end of the array is reached.

Example

```
s:dim i a [10]
s:a [0] = 'S'
s:a [1] = '-'
s:a [2] = 'B'
s:a [3] = 'a'
s:a [4] = 's'
s:a [5] = 'e'
s:a [6] = 0
s:print @stringize a [0]
S-Base
s:print @stringize (a [2], 2)
Ba
s:print @stringize a [2]
Base
```

@sum

Syntax

@sum (***variable*** | ***filefield*** [, ***length***])

Purpose

To sum all or part of an array.

Arguments

variable | ***filefield*** is a variable or file field (of array type) to sum. If a subscript is given, it is used as the starting element for the summation.

length is the number of elements to sum (all by default).

Returns

Integer or float (same type as ***variable*** | ***filefield***).

@sum returns the sum calculated from the array elements.

Use

@sum can be used with integer or float variables and file fields. It does a rapid summation of the specified range (starting element and length) within the array.

Example

Given an array 'x' with 10 elements containing the values 0 to 9:

```
s:print @sum (x)
45
```

```
s:print @sum (x[5])
35
s:print @sum (x,5)
10
s:print @sum (x[7],1)
7
```

Given a field 'plant.price' which is an array of five floats:

```
| sum up all the prices
print "Price total is ",@sum (plant.price)
```

@tan

Syntax

@tan ***number***

Purpose

To calculate the tangent of a number.

Arguments

number is the number in radians to use in the calculation.

Returns

Float.

@tan returns the tangent of ***number***.

@tanh

Syntax

@tanh ***number***

Purpose

To calculate the hyperbolic tangent of a number.

Arguments

number is the number in radians to use in the calculation.

Returns

Float.

@tanh returns the hyperbolic tangent of ***number***.

@teval

Syntax

@teval ***string***

Purpose

To evaluate a tokenised expression.

Arguments

string contains the tokenised expression to evaluate.

Returns

Type depends on expression.

@teval returns the result obtained from evaluating the tokenised expression in ***string***. If an error occurs during evaluation, the value is undefined.

Use

@teval is useful for evaluating expressions typed in by the user, and for calling procedures named at runtime within S-Base. You can convert a normal expression into a tokenised expression using **@token**. **@teval** is faster than **@eval**.

@teval can cause an error if asked to evaluate an invalid expression. It is common to set the error state to two before calling this function.

Example

```
s:a="@sqr 49"
s:print @eval a
7.00
s:b=@token a
s:print @teval b
7.00
```

See also

@eval

@time

Syntax

@time

Purpose

To get the current time.

Arguments

None.

Returns

Time.

@time returns the current time, read from the system clock.

See also

@date

@timedm

Syntax

@timedm ***timedate***

Purpose

To convert a time into a day/month string.

Arguments

timedate is the time or date to convert.

Returns

String.

@timedm returns a string representation of the day and month of ***timedate*** in the format dd/mm.

Example

```
s:print @timedm @timesn "21/12/89"
21/12
s:print @timedm @date
27/05
```

See also

@timemy, **@timesn**

@timedom

Syntax

@timedom ***timedate***

Purpose

To determine the day of the month from a time.

Arguments

timedate is the time or date to check.

Returns

Integer.

@timedom returns the day of the month from ***timedate*** (1-31)

Example

```
s:print @timedom @timest "15/11/89.12:34:05"
15
s:print @timedom @time
27
```

@timedow

Syntax

@timedow ***timedate***

Purpose

To determine the day of the week from a time.

Arguments

timedate is the time or date to check.

Returns

Integer.

@timedow returns the day of the week from ***timedate*** (Sunday is 0, Monday is 1, ..., Saturday is 7).

Example

```
s:print @timedow @timesn "01/06/89.12:34:05"
4
```

@timedoy

Syntax

@timedoy ***timedate***

Purpose

To determine the day of the year from a time.

Arguments

timedate is the time or date to check.

Returns

Integer.

@timedoy returns the day of the year from ***timedate*** (0-365, 1st Jan = 0).

Example

```
s:print @timedoy @timest "6/8/89.12:34:05"
217
```

@timefod

Syntax

@timefod ***timedate***

Purpose

To calculate the time at the start of the day.

Arguments

timedate is the time or date to convert.

Returns

Time or date (same as ***timedate***).

@timefod returns the time or date at the start of the day specified by ***timedate***, i.e. the time at 12:00:00am on the same day. It is only useful for time types.

Example

```
s:print @timefod @timest "12/5/89.4:55"
12/05/89.00:00:00
```

See also

@timefom, **@timefoy**

@timefom

Syntax

@timefom ***timedate***

Purpose

To calculate the time at the start of the month.

Arguments

timedate is the time or date to convert.

Returns

Time or date (same as ***timedate***).

@timefom returns the time at the start of the month specified by ***timedate***, i.e. the time at 12:00:00am on the first of the month.

Example

```
s:print @timefom @timesn "12/5/89":/n
01/05/89
s:print @timefom @date
01/05/1992
```

See also

@timefod, **@timefoy**

@timeform

Syntax

@timeform (***format_string***, ***time***)

Purpose

To format a time according to a format string.

Arguments

format_string is the format to use.

time is the time to format.

Returns

String.

@timeform returns the formatted time as a string.

Use

@timeform is useful when the standard time formatting options (like :/n and :/t) are inadequate. S-Base basically returns ***format_string***, but replaces some special fields with values from ***time***.

The following special fields are available to place within the format string:

Command	Replaced by
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	standard date and time string
%d	day of month as a number (1-31)
%H	hour on 24-hour clock (0-23)
%I	hour on 12-hour clock (1-12)
%j	day of year as a number (1-366)
%m	month as a number (1-12)
%M	minute (0-59)
%p	AM or PM
%S	second (0-59)
%U	week of year (0-52), Sunday is first day
%w	weekday as a number (0-6, Sunday is 0)
%W	week of year (0-52), Monday is first day
%x	standard date string
%X	standard time string
%y	year within century (0-99)
%Y	year (1970-?)
%Z	time zone name
%%	a percent sign (%)

Example

```
@timeform ("It's %A, %d %B %Y", @timesn "30/5/89")
| This will return "It's Tuesday, 30 May 1989"
```

@timefoy

Syntax

@timefoy ***timedate***

Purpose

To calculate the time or date at the start of the year.

Arguments

timedate is the time or date to convert.

Returns

Time or date (same as ***timedate***).

@timefoy returns the time at the start of the year specified by ***timedate***, i.e. the time at 12:00:00am on 1 January of that year.

Example

```
s:print @timefoy @timesn "12/5/89":/n
01/01/89
```

See also

@timefod, @timefom

@timeh

Syntax

@timeh ***timedate***

Purpose

To determine the hour from a time or date.

Arguments

timedate is the time or date to check.

Returns

Integer.

@timeh returns the hour from ***timedate*** using the 24 hour clock (0-23). It is only useful for times, since a date will always return 0.

Example

```
s:print @timeh @timest "15/11/89.12:34:05"
12
```

@timemoy

Syntax

@timemoy ***timedate***

Purpose

To determine the month of the year from a time or date.

Arguments

timedate is the time to check.

Returns

Integer.

@timemoy returns the month of the year from ***timedate*** (0 to 11, January is 0).

Example

```
s:print @timemoy @timest "01/06/89.12:34:05"
5
```

@timemy

Syntax

@timemy ***timedate***

Purpose

To convert a time or date into a month/year string.

Arguments

timedate is the time or date to convert.

Returns

String.

@timemy returns a string representation of the month and year of ***timedate*** in the format mm/yy.

Example

```
s:print @timemy @timesn "21/12/89"
12/89
```

@times

Syntax

@times ***time***

Purpose

To determine the second from a time.

Arguments

time is the time to check.

Returns

Integer.

@times returns the current second from ***time*** (0-59).

Example

```
s:print @times @timest "01/06/89.12:34:05"
5
```

@timesi

Syntax

@timesi ***string***

Purpose

To convert a string representation of a time interval to a time.

Arguments

string is the string to convert.

Returns

Time.

@timesi returns the time interval represented by the string.

Example

```
s:print @int @timesi "1s"
1
s:print @int @timesi "1u1s"
61
s:print @int @timesi "1h1u1s"
3661
```

@timesn

Syntax

@timesn ***string***

Purpose

To convert a string in natural format to a time.

Arguments

string is the string to convert.

Returns

Time.

@timesn returns the time representation of ***string***.

Use

@timesn is useful for converting user input to internal values. S-Base uses this function to process dates entered by the user.

For trailing parts of the date not specified, S-Base uses the current date.

Example

```
! Assuming today is 18 May 1992:
s:print @timesn "30/12/89":/n
30/12/89
s:print @timesn "12":/n
12/05/92
s:print @timesn "12/6":/n
12/06/92
```

@timest

Syntax

@timest ***string***

Purpose

To convert a string in time format to a time.

Arguments

string is the string to convert.

Returns

Time.

@timest returns the time denoted by string.

Example

```
s:print @timest "30/12/89,12:56:08" + @totime 1
30/12/89,12:56:09
```

@timeu

Syntax

@timeu ***timedate***

Purpose

To determine the minute from a time or date.

Arguments

timedate is the time or date to check.

Returns

Integer.

@timeu returns the current minute from ***timedate*** (0-59).

Use

This function is only useful with times since it will always return 0 when given a date argument.

Example

```
s:print @timeu @timest "01/06/89.12:34:05"
34
```

@timey

Syntax

@timey ***timedate***

Purpose

To determine the year from a time or date.

Arguments

timedate is the time or date to check.

Returns

Integer.

@timey returns the year from ***timedate***.

Example

```
s:print @timey @timest "01/06/89.12:34:05"
1989
```

@todate

Syntax

@todate ***value***

Purpose

To convert a value to a date.

Arguments

value is the value to convert, which can be an integer, time or float value.

Returns

Date.

@todate returns the date equivalent of ***value***.

Use

This function is useful when a number must be converted into a date for processing. ***value*** is first converted to an integer then to a date so that it represents the number of days since the first day of the year 0AD (assuming there was such a year).

Example

```
s:print @date
27/05/92
s:print @date + @todate 1
28/05/92
```

@tof

Syntax

@tof (*file* [, *index*])

Purpose

To find the first record in a file according to the given index ('top of file').

Arguments

file is the file to search.

index is the index (label or number) to use for the search. If none is given, record number order is assumed.

Returns

Integer.

@tof returns the record number of the first record in the file, or 0 if the file is empty.

Use

When no index is given, **@tof** will only ever return the first record in the file (1), or 0 if the file is empty.

See also

@bof, **@next**, **@previous**

@token

Syntax

@token *string*

Purpose

To tokenise a string.

Arguments

string is the string to tokenise.

Returns

String.

@token returns the tokenised string.

Use

@token is useful in conjunction with **@teval**, which evaluates tokenised expressions. Where an expression is to be used many times — in a search, for example — it is better to tokenise the expression once at the start and use **@teval** to evaluate it, than to use **@eval** to tokenise and evaluate the string each time.

Example

```
s:print @teval @token "6*6+6"
```

```
42
```

See also

@teval

@tolower

Syntax

@tolower ***string***

Purpose

To convert a string to lower case.

Arguments

string is the string to convert.

Returns

String.

@tolower returns a lower-case version of ***string***.

Example

```
s:print @tolower "MARKET"
market
s:print @tolower "market!!!"
market!!!
```

@totime

Syntax

@totime ***value***

Purpose

To convert a value to a time.

Arguments

value is the value to convert, which can be an integer, date or float value.

Returns

Time.

@totime returns the time equivalent of ***value***.

Use

This function is useful when a number must be converted into a date for processing. ***value*** is first converted to integer then to a time so that it represents the number of seconds since the first second of the first day of the year 1970.

Example

```
s:x = @time
s:print x
```

```
27/05/92,22:11:43
s:print x + @totime 60
27/05/92,22:12:43
```

@toupper

Syntax

@toupper ***string***

Purpose

To convert a string to upper case.

Arguments

string is string to convert.

Returns

String.

@toupper returns a upper-case version of ***string***.

Example

```
s:print @toupper "Market"
MARKET
s:print @toupper "A5000 Computer"
A5000 COMPUTER
```

@trim

Syntax

@trim ***string***

Purpose

To remove trailing spaces from a string.

Arguments

string is the string to process.

Returns

String.

@trim returns ***string*** with trailing spaces removed.

Use

@trim is useful for removing trailing spaces from user input before processing.

Example

This is a keypress handler for a field where the user should enter a video title. It has one field ('find_string').

```
if @justaltered
| search for the title as entered by the user
   rec = @findn (update title, "title", @trim
```

```
find_string)
   if rec
| title exists, so display in value bar
      card update value @handle, title.title
   endif
endif
```

@val

Syntax

@val ***string***

Purpose

To convert a string to a number.

Arguments

string is the string to convert.

Returns

Integer or float.

@val returns the converted number, whose type depends on the string being converted.

Use

@val converts as many characters as possible to a number, including those after the decimal point, if present. Conversion stops at the first non-numeric character.

@val decides whether to return an integer or float number on the basis of the presence or absence of a decimal point in the string.

Example

```
s:print @val "5"
5
s:print @val "-6.34asd"
-6.34
s:print @val "fred"
0
```

@version

Syntax

@version

Purpose

To obtain the version of S-Base running the application.

Returns

Integer.

@version returns the S-Base version number multiplied by 100.

Use

@version allows you to check that the current version of S-Base is capable of running your application. You can do this check when starting up the application.

Version 1.0 of S-Base returns 100 from this function.

@vrecsize

Syntax

@vrecsize (***file*** [, ***count***])

Purpose

To calculate the size of a record in a variable-length file.

Arguments

file is the file structure to use for the calculation.

count is the number of items in the transaction. By default this is the value of the file's *variable length field.*

Returns

Integer.

@vrecsize returns the size in bytes of the variable-length record needed to hold a record of structure ***file*** with ***count*** items in it.

Use

@vrecsize is often used with *transaction files*. Usually, a transaction file is implemented with a fixed-length header which may be followed by a number of transaction *items*.

The header contains information that all transactions have: the transaction type, date, customer, number of items, etc.

The transactions items are an array, the last field in the transaction record. In transactions with no items (where `tran.num_items` = 0), S-Base will simply save the transaction header without any items. The length of this record is `@vrecsize (tran, 0)`. The length of any record is **@vrecsize** (***file***, ***count***). Generally ***count*** will not be required, as S-Base knows which field specifies the number of items in the record and uses this as the item count.

@vrecsize can be used to find out the true size of a record in a variable length file.

Example

Assume a 'tran' file with these fields: tran.type, tran.date, tran.num_items, tran.customer and tran.item [100] of type 'item'. The variable length field is as usual the last field (tran.item) and the number of items is given by the 'tran.num_items' field.

Assume 'item' structure with these fields: item.code, item.quantity and item.value.

```
| Save a transaction to disc:

print @vrecsize tran, " bytes to save."
create tran

| Load a transaction from disc
load tran, trannum
print "With 1 more item, transaction would be";
print @vrecsize (tran, tran.num_items + 1), " bytes."
```

See also

@recsize

Appendix A: Key codes

The following table lists the key codes returned by the wimp when a key is pressed. The function **@key** returns these values. Keypress codes between 32 and 255 correspond to the extended ASCII codes used under RISC OS, and you can convert between the key code and the ASCII character using **@chr** and **@asc**. Characters between 128 and 255 inclusive are not listed as these depend on the font used to display them.

Decimal	Hex &	Character
0	0	ctl-@
1	1	ctl-A
2	2	ctl-B
3	3	ctl-C
4	4	ctl-D
5	5	ctl-E
6	6	ctl-F
7	7	ctl-G
8	8	ctl-H
9	9	ctl-I
10	A	ctl-J
11	B	ctl-K
12	C	ctl-L
13	D	return/ctl-M
14	E	ctl-N
15	F	ctl-O
16	10	ctl-P
17	11	ctl-Q
18	12	ctl-R
19	13	ctl-S
20	14	ctl-T
21	15	ctl-U

Decimal	Hex &	Character
22	16	ctl-V
23	17	ctl-W
24	18	ctl-X
25	19	ctl-Y
26	1A	ctl-Z
27	1B	ctl-[(escape)
28	1C	ctl-\
29	1D	ctl-]
30	1E	ctl-^
31	1F	ctl-_
32	20	space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+

Decimal	Hex &	Character
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G

Decimal	Hex &	Character
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	£
97	61	a
98	62	b
99	63	c

Decimal	Hex &	Character
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	delete back

Decimal	Hex &	Character
384	180	Print
385	181	f1
386	182	f2
387	183	f3
388	184	f4
389	185	f5
390	186	f6
391	187	f7
392	188	f8
393	189	f9
394	18A	tab
395	18B	copy
396	18C	left arrow
397	18D	right arrow
398	18E	down arrow
399	18F	up arrow
400	190	sh-Print
401	191	sh-f1
402	192	sh-f2
403	193	sh-f3
404	194	sh-f4
405	195	sh-f5
406	196	sh-f6
407	197	sh-f7
408	198	sh-f8
409	199	sh-f9
410	19A	sh-tab
411	19B	sh-copy

Decimal	Hex &	Character
412	19C	sh-left arrow
413	19D	sh-right arrow
414	19E	sh-down arrow
415	19F	sh-up arrow
416	1A0	ctl-Print
417	1A1	ctl-f1
418	1A2	ctl-f2
419	1A3	ctl-f3
420	1A4	ctl-f4
421	1A5	ctl-f5
422	1A6	ctl-f6
423	1A7	ctl-f7
424	1A8	ctl-f8
425	1A9	ctl-f9
426	1AA	ctl-tab
427	1AB	ctl-copy
428	1AC	ctl-l arrow
429	1AD	ctl-r arrow
430	1AE	ctl-d arrow
431	1AF	ctl-u arrow
432	1B0	ctl-sh-Print
433	1B1	ctl-sh-f1
434	1B2	ctl-sh-f2
435	1B3	ctl-sh-f3
436	1B4	ctl-sh-f4
437	1B5	ctl-sh-f5
438	1B6	ctl-sh-f6
439	1B7	ctl-sh-f7

Decimal	Hex &	Character
440	1B8	ctl-sh-f8
441	1B9	ctl-sh-f9
442	1BA	ctl-sh-tab
443	1BB	ctl-sh-copy
444	1BC	ctl-sh-l arrow
445	1BD	ctl-sh-r arrow
446	1BE	ctl-sh-d arrow
447	1BF	ctl-sh-u arrow
458	1CA	f10
459	1CB	f11
460	1CC	f12
461	1CD	insert
474	1DA	sh-f10
475	1DB	sh-f11
476	1DC	sh-f12
476	1DD	sh-insert
490	1EA	ctl-f10
491	1EB	ctl-f11
492	1EC	ctl-f12
493	1ED	ctl-insert
506	1FA	ctl-sh-f10
507	1FB	ctl-sh-f11
508	1FC	ctl-sh-f12
509	1FD	ctl-sh-insert

Appendix B: Wimp event codes

The following event codes are defined in the Window Manager, and may be passed to raw event handlers. See the RISC OS Programmers' Reference Manual for more details.

Code	Name	Meaning
0	Null	Returned when no other event can be returned. This event indicates the machine is momentarily idle.
1	Redraw	A window should be redrawn.
2	Open	A window should be opened.
3	Close	A window should be closed.
4	Ptr-leave	The pointer is leaving a window.
5	Ptr-enter	The pointer is entering a window.
6	Click	The user has clicked the mouse in a window.
7	Drag box	The user has finished dragging something.
8	Keypress	The user has pressed a key.
9	Menu select	The user has selected a menu option.
10	Scroll	A window should be scrolled.
11	Lose caret	The caret has left a window.
12	Gain caret	The caret has entered a window.
13	Pollword	The application's poll word has become non-zero.
14-16	-	Not used.
17	Message	A message from another task.
18	R. Message	A recorded message from another task (requires a response).
19	Msg. Ack.	Recorded message was not acknowledged.

Index

A

Absolute value 149
Arc cosine 149
Arc sine 153
Arc tangent 153
Arrays 8-9, 53
Arrays, searching 181
Arrays, sorting 115
Arrays, summing 223
ASCII 152, 158
Attaching tables to cards 132
Automatic recalculation 33, 131
Automatic report generation 29, 101, 129

B

Bitstrings 12, 154, 212, 218
Buffers 17
Buffers, size of 177

C

Cards 25
Cards, checking if altered 151
Cards, element positions 23
Command line 80
Control structures 43, 62-63, 65, 145
Cosine 162
Currency 165
Cursor position 162

D

Data export 7, 13, 88
Data import 66, 154
Date 163
Dates, converting to 234
Dates, formatting 221
Degrees 164
Deleted records, finding 165
Deleted records, removing 81
Disc heaps 4

E

Element types 26, 124
Entries, number of 173
Entry size 98, 127
Error handling 48, 57-58, 175
Evaluating expressions 176
Evaluation and field width 29, 129
Execution, stopping 59
Exiting applications 59
Exponent 177
Expressions, evaluating 176, 224
Expressions, tokenising 235
External file fields 4
Externals 115, 178
Externals, loading into buffers 16
Externals, saving from buffers 19

F

File field 4
File fields 60, 179
File locking 141, 194
File, end of 174
File, size of 177
Files, closing 46
Files, current position in 213
Files, last record 156
Files, opening 206
Files, reindexing 92
Files, zapping 146
Files, zeroing memory buffer 147
Filespec 3
Filters 107
Floating structures 142
Flushing data 61
Format areas 27, 96, 125
Formatting values 221

G

Graphics 44, 46, 55, 64, 78-79, 81, 87, 91, 211

H

Handlers 27, 127
Handles 3
Hourglass 51

I

Icons 31
Indexes, loading 193
Indexes, number of 186
Indexes, rebuilding 92
Indexes, zapping 146
Inserting and deleting entries 133

K

Keyboard, scanning 187
Keypress 191

L

Language file 202
Logarithm 193, 195

M

Maximum 197
Mean 197
Memory 157, 185
Memory access 185, 189
Menu, current 198
Menu, handle 198
Menu, selected item 199
Menus 74
Menus, fading 77
Menus, text 173
Menus, ticking 78
Message file 202
Minimum 201
Mouse clicks 158
Moving the cursor 35, 131
Multi-entry reports 100
Multiuser support 141, 194

N

Nested cards 28
Nested tables 129
Numbers, formating 221
Numbers, rounding 217

O

Objects 3
Ordering 208
OS units 5
OSCLI 80

P

Page number 210
Polling 4
Procedures 49, 90
Program menus 84
Programs, running 104
Protected elements 35, 131

R

Radians 214
Random numbers 216
Record buffers 5
Record locking 141, 194
Record number ordering 108
Record number, current 214
Records, creating 47
Records, deleted 165
Records, deleting 52
Records, loading 67
Records, next available 205
Records, number of 215
Records, saving 105
Records, size 239
Records, size of 215
Records, updating 142
Records, variable length 239
Redrawing and the cursor 131
Reports 95
RISC OS buffers 151

RISC OS command line 80
RISC OS strings 222
Root 220
Rounding 217

S

S-Base version 238
Searching 106, 179, 183, 204, 208, 211, 235
Selectors 30, 111
Selectors and Icons 129
Selectors, fading 212
Selectors, selected entries 218
Selectors, text 173
Sign 219
Sine 219
Square root 220
Standard deviation 217
Strings 192, 200, 216, 220-221, 236-237
Strings, converting to numbers 238
Strings, formatting 184, 203, 205, 209
Strings, from arrays 222
Strings, length 192
Strings, matching 196
Strings, searching 188
Strings, trimming spaces 237
SWIs 117
System calls 117

T

Table entries 126
Table extension 133
Table, number of entries 177
Tables 31, 123
Tangent 224
Text only reports 99
The card layout 25
The order of format elements 27, 99, 126
The table layout 123
The value bar 34, 131
The Window Manager 4
Time 225

Time, elapsed 202
Time, formating 228
Times, converting to 236
Times, formatting 221
Tokenising 4, 235
Translation 202
Truncation 188

U

Updating 108, 180
User control 32, 100, 129

V

Variables, address of 149
Variables, clearing 45
Variables, listing 70
Variables, local 68
Vdu 144
Version 238

Longman Logotron

124 Cambridge Science Park
Milton Road
Cambridge CB4 4ZS

Tel. (0223) 425558
Fax (0223) 425349